

U *S I N G*

*TURBO
PROLOG*

SECOND EDITION

Kelly M. Rich &
Phillip R. Robinson



74679

BORLAND-OSBORNE/McGRAW-HILL

PROGRAMMING SERIES

BORLAND-OSBORNE

Using Turbo Prolog®

AN (g) Y
OR
OR (g) 0
IF (:-) 81

Kelly Rich
and
Phillip R. Robinson

B.4
INC. Mec. Ele

BORLAND-OSBORNE/McGRAW-HILL

B U S I N E S S S E R I E S

✓
Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne McGraw-Hill at the above address.

A complete list of trademarks appears on page 351.

Copyright C 1988 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 8988

ISBN 0-07-881302-6

Information has been obtained by Osborne McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Osborne McGraw-Hill, or others, Osborne McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information.

I wish to dedicate this book to Micki and Kenny, because without showing me the strength they have for their projects, I never would have had the strength to complete mine.

Kelly Rich

Contents

Foreword **xi**

Introduction **xiii**

Part One 1

1 Background and Setting Up 3

Prolog's Strengths 4

What You Need To Run Turbo Prolog 7

Installing Turbo Prolog 9

The Toolbox 11

Starting Prolog 11

2 The Turbo Environment: Menus and Windows 13

Starting the Program 14

Windows 16

Menu Bar 21

The ESC Key 21

Hot Keys 37

3 The Editor 39

Selecting the Editor 40

Leaving the Editor 41

Saving a File 42

Help in the Editor	42
Function Keys	43
Cursor Movement Commands	44
Inserting and Deleting Text	46
Block Functions	48
Search and Replace	48
WordStar-like Block Functions	50
Miscellaneous Commands	51
Hot Keys	52

Part Two 53

4	<i>Facts: Objects and Relationships</i>	55
	Procedural Versus Declarative	55
	Objects and Relationships	56
	Syntax	58
	Facts	59
	Comments	61
	Program Divisions	62
	The First Program Example	64
5	<i>The Basics of Programming in Prolog</i>	75
	Prolog's Building Blocks	77
	Prolog's Matching Mechanism	80
	Compound Goals and Rules	84
	The Search for Solutions	90
	Using Not()	98
	Pressing Onward!	99
6	<i>The Sections of a Turbo Prolog Program</i>	101
	Clauses Section	102
	Predicates Section	102
	Domains Section	108
	Goal Section	114
	Advanced Program Sections	116
	Advanced Declarations	119
	List of Turbo Prolog Program Sections	123

7	<i>Controlling Prolog's Search</i>	125
	Control Structures	126
	Prolog's Control Predicates	128
	Cutting Down on Search Time	132
	Prolog Clauses Versus Procedural Statements	140
8	<i>Prolog Math</i>	143
	Numbers as Symbols	144
	Evaluating Expressions	147
	Mathematical Functions	150
	Adding a Random Flare	153
	Comparing Prolog Terms	155
	<i>Part Three</i>	159
9	<i>Working with Lists and Recursion</i>	161
	The List Data Structure	163
	Unification and Lists	166
	Recursion	168
	Assigning Values in a Recursive Process	174
	Creating Lists	176
	Building Lists Dynamically	179
10	<i>Input and Output with Turbo Prolog</i>	183
	Working with Files	183
	Tail Recursion Optimization	189
	Working with Strings	191
	A Few More File-handling Predicates	202
	Seeing It All Work Together	203
11	<i>Knowledge Bases in Turbo Prolog</i>	207
	Data Representation	208
	Internal Databases	209
	Multiple Internal Databases	220
	Using the Database to Store Values	223
	External Databases	225
	B+ Trees	233

Part Four 241

12	<i>Advanced Techniques</i>	243
	Advanced Compiler Features	243
	Coding Techniques	261
13	<i>Using the BGI and the Toolbox</i>	281
	The BGI	282
	Turbo Prolog Toolbox	302
	Sound	305
	Readkey() Predicate	306
14	<i>An Advanced Application</i>	311
	Getting Started	312
	Glancing at the Code	313
	Supporting Predicates	314
	Bringing in the Data	315
	Calculating the Intervals	317
	Writing Out the Results	324
A	<i>Built-in Predicates</i>	335
	<i>Index</i>	353

Foreword

Borland International launched the first version of Turbo Prolog nearly three years ago. Now, in conjunction with the release of Turbo Prolog 2.0, I am pleased to introduce the second edition of *Using Turbo Prolog* from Osborne/McGraw-Hill. Written by Kelly Rich and Phillip R. Robinson, this book encompasses the many features found in the first version of Turbo Prolog and fully addresses all the new capabilities of version 2.0.

Prolog is the premier language used for artificial intelligence research and has been used for years to implement expert systems and natural language interfaces. With the introduction of Turbo Prolog, the power of Prolog became accessible to PC users. The additions in Turbo Prolog 2.0 of the Borland Graphics Interface (BGI), external databases, B+ trees, EMS support, exception handling and error trapping makes this the ideal language environment for all AI applications. Turbo Prolog is also a powerful developmental tool for general applications.

Using Turbo Prolog is the best introductory Turbo Prolog book on the market because no one can know the problems and pitfalls of learning Turbo Prolog better than Kelly Rich. Through his work in Borland's Technical Support Department, Kelly has learned which aspects of the language are most likely to stump or confuse new users. And, because he knows the language so well, he can carefully and logically lead users to a clear and useful understanding of the Turbo Prolog environment.

Using Turbo Prolog

Phillip Robinson is also a veteran Turbo Prolog programmer, and the author of the first edition of this book. In addition to writing several computer books, Phillip is the former West Coast editor of *Byte* magazine, and his articles have appeared in many other computer magazines.

I confidently recommend *Using Turbo Prolog* because learning from the pros is the best way to learn a language. This book gives readers a firm foothold in the world of Prolog and shares those inside hints and tips that only veteran users of Turbo Prolog can know.

A handwritten signature in black ink, appearing to read 'PKah', with a long horizontal stroke extending to the right.

Philippe Kahn
President
Borland International, Inc.

Introduction

When communicating, it is important that the idea being communicated moves accurately from the source to the target. This rule is true for both natural, spoken languages and languages that are used for “talking” to computers. In a programming language, you (the programmer) are the source, and the language compiler is the target. Whether you are programming in BASIC, C, or Prolog, care must be taken to write a program that the compiler can properly digest. As well, you must be sure that the compiler can correctly translate the meaning of the code you have written in order to produce a working, computer program.

Computer programming is much like holding a conversation with a friend: You must take care that your ideas are getting across correctly. For communication to be successful, your choice of words must be a part of the language being spoken, and your sentences must conform to the grammar of the language being used. If you deviate from these two “rules” of speaking, the ideas you wish to convey may become muddled in the communication and a misunderstanding may occur.

By the very definition of the word *communication*, the conversation between you and your friend must be equally understood by both parties. Not only do the sentences you speak need to conform to the rules of the language, but the meaning of the sentence spoken must also be decipherable. When both of these restrictions

are overcome, then communication between the source and the target can be successful.

In computer programming, conforming to the rules (grammar) of the language is known as following the syntax of the language. When, in programming, you create a "sentence" that the compiler cannot translate, it is known as making a syntax error.

On the other hand, it is possible to create a correct sentence, but have the meaning of the sentence be misconstrued when the program is run. In these cases, the compiler is able to create a working program out of the code you have written, but the meaning of the code does not translate logically into a working program. This type of error is known as a logical error; the "semantics" (or meaning) of the code do not translate properly. A semantic error will prohibit your program from coming up with the correct results, even though your program appears to run correctly.

As with the above analogy, the programs you write in Turbo Prolog must conform to Turbo Prolog's rules of communication. In this book, care is taken to show you the correct syntax for the Turbo Prolog language. But, unlike many other introductory programming books, *Using Turbo Prolog* goes beyond the syntax of the language and shows you how to put together programs that make sense logically.

While it is not intended to replace the manuals provided with the Turbo Prolog compiler, *Using Turbo Prolog* will supplement the tutorials that are given in the Turbo Prolog owner's manuals. Going beyond the syntax of the Turbo Prolog language, *Using Turbo Prolog* concentrates on teaching you how to put the language to work.

Providing an in-depth coverage of the logical problems you will encounter, *Using Turbo Prolog* covers areas such as looping structures, data organization, advanced techniques, and many areas not considered in most common language tutorials. Simply picking up a language's syntax can be obtained from almost any programming tutorial. Learning how to use this syntax (by showing solutions to the problems that are most often encountered) is the area that is emphasized in this book.

The book begins with an introductory section. First, Prolog as a language, is introduced. From here, the Turbo Prolog environment

and editor are covered in detail, then a short guide follows explaining how Turbo Prolog's programs are organized.

The next section provides a more thorough introduction to the language and its syntax. In Chapter 5, the workings of the language are covered in detail including backtracking, Turbo Prolog's unique tracing facility. Chapter 6 covers the sections of a Turbo Prolog program, showing how declarations are organized in conjunction with the main body of the program code.

In Chapter 7, control structures in Prolog are introduced. Often missed in other Prolog tutorials, this chapter details many of the procedural programming techniques that need to be used when creating programs in Prolog.

The last chapter in this section describes how Turbo Prolog uses math. A clear and easy-to-understand discussion is given on how to incorporate Turbo Prolog's mathematical functions, conditional operators, and the equality predicate.

The third section of the book covers the advanced features found in the compiler itself. Using Turbo Prolog's powerful windowing predicates, directing the input and output, and setting up databases are the main topics of the chapters found in this section. Also, an introduction is given to the B+ tree structures that are used to index large databases.

The last section of the book features advanced programming techniques. Starting with a section on creating and debugging larger programs, Chapter 12 continues by describing how to tackle more difficult programming problems. In this section, detail is given on how to use backtracking to your advantage and how to control the flow through the programs that you create. Advanced looping structures, nested loops, and advanced debugging methods are also presented.

In the next chapter, the Borland Graphics Interface is introduced along with a section on how the tools in the Turbo Prolog Toolbox can be added to your programs. Using the ideas presented in Chapter 13, your Prolog programs can be given a professional flair by designing high-level user interfaces and quality graphics to go along with the displays that your programs produce.

In the last chapter, an advanced application is shown, and its processing is described in detail. Here, you can see how the tech-

niques described throughout the book can be put to use in a fully functional program. This program will give you ideas on how advanced techniques can be added to your own programs, and shows how experienced Prolog programmers structure their code.

In addition to the tutorial chapters, an appendix is given that documents all of the built-in predicates of Turbo Prolog version 2.0. Next to each predicate, a list of the usable flow patterns and argument types is given, providing an indispensable reference tool.

Because *Using Turbo Prolog* moves from showing the syntax of the language to detailing how to use these features, this book will educate both novice and intermediate programmers. The chapters describe how to *use* Turbo Prolog, rather than just giving a description of how to form correct clauses that the compiler can understand. Instead of just learning the language's grammar, *Using Turbo Prolog* teaches you how to converse in the most powerful fifth-generation language available for personal computers.

—Kelly Rich

Part One

Part One provides you with a basic introduction to Turbo Prolog by focusing on the tools you need to begin programming and on the unique characteristics of the Turbo Prolog environment. This section also describes the powerful Turbo Prolog Editor that allows you to write and debug programs quickly and efficiently.

1 Background and Setting Up

After its invention in the early 1970s, Prolog quickly became the leading artificial intelligence language in European laboratories and universities. By the late 1970s, versions of Prolog for microcomputers started to appear. One of the most popular microcomputer Prolog compilers was micro-Prolog, and many other Prolog books are devoted to it. But micro-Prolog does not offer the wealth of predicates that you'll find in Turbo Prolog. What's more, micro-Prolog is much slower than Turbo Prolog, in that a micro-Prolog program takes far longer to make a logical decision than does a similar Turbo Prolog program.

There wasn't a lot of interest in Prolog until Japanese computer scientists launched their famous "fifth-generation" project with the goal of designing new computers and software that would reign supreme into the 1990s and beyond. It wasn't simply coincidence that the language they chose to base their work on was Prolog. Suddenly, people began taking another look at Prolog and its capabilities.

Prolog's Strengths

Individual computer languages are rarely good for all types of problems. FORTRAN (short for "FORmula TRANslation"), for example, is used primarily by scientists and engineers, while COBOL (short for "COmmon Business Oriented Language") is used mainly in the world of accounting and business management. Each has functions, commands, and a style that tailors it more to one job than to another.

Prolog is designed to handle *logical problems*—problems for which decisions need to be made in an orderly fashion. Prolog tries to make the computer "reason" its way to a solution. It is particularly well suited to several types of AI problems. The two most significant of these are expert systems and natural-language processing.

Expert Systems

Expert systems are computer programs that imitate a human expert. They contain information (that is, a *database*) and a tool for understanding questions and finding the right answers to those questions in the database (that is, an *inference engine*). Typical applications for expert systems include medicine (where computers are asked to diagnose diseases) and geology (where computers are asked where oil might be found).

Turbo Prolog has a built-in structure for creating databases and has a ready-to-run inference engine. All you have to do is tell the program the rules to run by, and it will discover its own path to the appropriate information. Many other programming languages require that you write the rules, explain the rules, specify the paths, and generally work at a much more detailed level to create the inference engine before you ever begin to retrieve the needed information. Also, Turbo Prolog's database is built with the same structures that are used to write the rules—learn one facet of the problem and you know how to deal with both.

There is still a science and quite a bit of art to writing a program, even in Prolog where part of the work is done for you. You

need to specify how a program will read input information and write output information (such as graphics, sound, and file handling). And the order in which you set up the information can have an enormous impact on the speed with which the program finds correct answers.

Natural-Language Processing

Natural-language processing is the technique of forcing computers to understand human languages. Scientists who study natural-language processing hope to create hardware and software that will allow you to type or speak commands such as “move the Turbo Prolog files to the Prolog object directory” and have the computer answer or follow your directions. Today you have to use a command such as “b: copy a:*.pro \prolog[v]” to meet that end. Turbo Prolog uses a database and an inference engine to divide human language into different parts and relationships and thus tries to “understand” the meaning of text or speech. The Turbo Prolog Toolbox even includes a set of “parsing” tools to show you how to get started in natural-language work.

Logic

Prolog depends on logical manipulations. As a computer language, it is not unique in this respect; a number of other languages exist in the general field of logic programming. Such languages work with propositional logic, also known as predicate logic or propositional calculus. For instance, the following statements are logical statements:

People with baseball shoes play baseball.

Person #1 has baseball shoes.

What might be inferred, figured, deduced, calculated, computed, or determined from those statements is that Person #1 plays baseball.

Turbo Prolog allows the computer to handle the inferring part. It is a *declarative language* that asks only that you declare rules and facts about a situation; the program then handles the inferences. Procedural languages, such as BASIC and Pascal, ask that you also create an algorithm for handling an inference. Turbo Prolog has a built-in inference engine that automatically hunts through facts and builds or tests logical conclusions. The problem just given might seem trivial, but if you had a technical database loaded with thousands of interrelated facts and rules, it would not be practical to hand that list to a human and ask for a quick answer.

Resolution and Unification

Mathematical rules of logic that can reduce textual statements to symbolic representations have been around for years. But in 1965, J. Alan Robinson invented the *resolution principle*, which showed how these representations could be cast in the proper form and given to a computer for analysis.

Resolution is an inference rule—it lets the computer tell what proposition follows logically from other propositions. Software that employs the resolution principle works with logical *clauses* (you'll see these throughout this book). It uses *unification* to match a goal with a relation by investigating the variable values that will allow a proper match.

If you want to know more about the theoretical basis of Prolog and about resolution and unification, the book *Programming in Prolog* (New York: Springer-Verlag New York, Inc., 1984), by W. F. Clocksin and C. S. Mellish, is an excellent source of information. While there are many good books on logic and computers, Chapter 10 in Clocksin and Mellish does a particularly good, Prolog-style job of describing predicate logic. In fact, Clocksin and Mellish is frequently referred to as the “bible” of Prolog—the book that sets the standard for Prolog rules and regulations.

Turbo Prolog Versus Other Prologs

Standard Prolog is a pretty spare language, sticking to logical rule structures and database constructions. Many implementations of Prolog even lack the ability to handle “number-crunching” and text-processing problems. Turbo Prolog contains those abilities and goes far beyond the old Prolog standard, adding input and output commands, graphics, and even sound features to make it a full-fledged language able to grasp most of the powers of a personal computer. For that reason, although the logical kernel of Turbo Prolog programs will closely resemble Prolog standards, the additional input and output sections will not be easily translated into other brands of Prolog. This is not unusual. Most computer languages differ substantially in input and output commands from one company’s version to another.

What You Need to Run Turbo Prolog

Both versions 1.1 and 2.0 of Turbo Prolog will run on almost any IBM PC or compatible computer with at least 384K of RAM, two floppy disk drives, and DOS 2.0 or greater. I urge you to get a system with at least 512K of RAM (and preferably 640K). That extra memory will make computing smoother, not only with Prolog but also with many other PC programs, and will only cost a few extra dollars. If you use the minimum of 384K, you won’t be able to load extra memory-resident programs such as SideKick.

Version 1.1 of Turbo Prolog came on two disks and fit handily on a two-floppy system, whereas version 2.0 comes with many more examples and other files, and cries out for a hard disk system. The original files come on three disks, but this is misleading. Many of these files are “compressed,” packed into a smaller amount of memory by a special “archiving” utility program. To make them usable, they must be decompressed, “unpacked” to normal size.

The installation batch files (mentioned below) handle this step automatically, but you can also do it yourself by using the UNPACK program. To learn how to use this program (which is on one of the Prolog disks), just type **unpack** without any following parameters, and press ENTER. The program will then explain itself.

A hard disk is not too expensive compared to the original price of a computer system, and will make a great difference in any computing you do. Hard disks not only hold many times as much information—programs, files, utilities—as floppy disks, but also transfer that information much faster, saving and loading files in a fraction of the time that a floppy needs. If you have a system with 3 1/2-inch floppy disk drives, the type that holds 720K or more per disk, you'll feel less cramped than with the older 5 1/4-inch, 360K floppies, but you still won't have the elbow room or speed of a hard disk.

Graphics

Traditional Prolog programs do not have any strong dependence on graphics, sticking mainly to text interaction between the computer and the user. Turbo Prolog offers a vigorous set of graphics modes and commands, and the Turbo Prolog Toolbox takes advantage of them to add specialized chart, menu, and screen-design features to the Prolog language. If you would like to take full advantage of these, a color graphics card and monitor will be a necessary addition to your hardware. If you can afford EGA or even VGA graphics, you'll be impressed by the resolution and clarity of these newer graphics standards.

Disks

If you're going to use a 5 1/4-inch, floppy-based machine, you'll need to format and label five blank disks on which to hold the working Prolog files. These are the label names you should use:

SYSTEM

HELP/ERROR MESSAGES

LIBRARY

BGI GRAPHICS/VERSION 2 EXAMPLES

TUTOR/SAMPLE PROGRAMS

In fact, you should have more blank disks than that so that you can make backup copies of your original Prolog disks before creating the working set of five disks. (Formatting and backing up disks are discussed in your DOS manual or in any DOS handbook. Use the DOS `FORMAT` and `BACKUP` or `DISKCOPY` commands.) The Turbo Prolog disks are not copy-protected.

If you're going to use a hard disk machine, make sure you have at least 1.8 megabytes of free disk space (use the `DIR` command to see what is on your disk and how much space is free).

README

On the original Prolog disks you'll find a program called `README.COM` and a `README` text file. The `README` file contains information too recent to be included in the printed manual, such as bugs, new features, changed commands, and manual typos. To inspect this data, run the `README.COM` program (just type `readme` and press `ENTER` from the DOS prompt). The program will automatically display the `README` file, indicating the key commands for moving around in the file and printing the text. Check the `README` file before you go any further.

Installing Turbo Prolog

The various files of Turbo Prolog (the compiler files, help files, system configuration files, error message files, example files, and so on) need to be in particular directories to be easily reached by the

main compiler system. The subdirectories and their contents (beneath the main Prolog directory) are

BGI	BGI graphics files
EXAMPLES	Example programs
ANSWERS	Answers to the tutorial example programs
PROGRAMS	Sophisticated example programs
CPINIT	Files needed to interface with Turbo C
VER2EXAM	Example programs specific to version 2.0

Setting up these directories and copying the files in version 2.0 is easy: Just use the installation batch files. If you have a hard disk system, first remove all the old version 1.0 and 1.1 Prolog system files (all except your own Prolog source-code files) from the hard disk. Then type **installh** and press ENTER. This will start the **INSTALLH.BAT** file, which contains an automatic installation sequence for putting Turbo Prolog on a hard disk.

If you have a floppy disk system, get your five blank and formatted disks ready, then type **install** to cue the **INSTALL.BAT** commands. Each of these installation batch files will display instructions on the screen.

Paths and Autoexecs

The last part of installing Turbo Prolog is preparing your DOS system files to accept the program. You should set aside the right amount of memory for Prolog by adding the lines

```
files=20
```

```
buffers=40
```

to the **CONFIG.SYS** file, which is read automatically each time you start your computer. (This file is explained in your DOS manual.) You can use a text editor such as the one found in Turbo Prolog to add these lines or create the file.

If you also add the line

```
c:\prolog
```

to the DOS path, you'll be able to start the Turbo Prolog compiler from inside any directory, just by typing **prolog**. You can set this up automatically each time you start your computer by adding the line

```
path=c:\prolog
```

to the AUTOEXEC.BAT file. (This file is explained in DOS manuals. It loads automatically any time you restart or reboot your computer.)

After you make these changes, you'll need to restart your computer (turn the power off or use the CTRL-ALT-DEL key combination) to put them in place.

The Toolbox

If you decide to buy and use the Turbo Prolog Toolbox with its repertoire of special menu, chart, graphics, communications, and other commands, you can copy its files to a Toolbox subdirectory under the Prolog directory. There is no installation file for this task, so you'll have to use the DOS MKDIR TOOLBOX command to create a new directory and the COPY command to move the files from the Toolbox disks to your hard disk. (If you're using a floppy-based system, it won't be practical to work with the five Turbo Prolog system disks plus two more Toolbox disks.)

Starting Prolog

The next chapter will handle how to start Prolog, but if you want to jump right in, just get into the Prolog directory and type **prolog**, then press ENTER. You'll see a copyright notice and a version

number display. If a file named WORK.PRO was in the Prolog directory, you'll see the main set of windows and the menu, and WORK.PRO will appear in the Editor window. Use the cursor-arrow keys to move around in the menu, press ENTER to select a menu option, and press ESC any time you want to return to this menu. Press ALT-H to see a list of *hot keys*: key combinations that let you execute commands without wading through the menus. You can use Quit in the Files menu, or the hot-key combination ALT-X, to leave Turbo Prolog and return to DOS. (Your system will need to be able to find the COMMAND.COM DOS file on the hard disk or on a floppy disk to successfully leave Prolog and get back to DOS.)

2 The Turbo Environment: Menus and Windows

Learning to use a particular computer language is not just a matter of learning the general theory of that language. You also have to master a particular implementation. Compare this to driving a car, where knowledge of the function of the steering wheel, the accelerator, and the brake pedal isn't enough. You don't really know your car until you've driven it under various conditions, learned the layout of its instruments and controls, gotten a feel for the gear shift, and have a place to get it fixed.

Computers and computer languages aren't nearly as standardized as cars. No two Prologs are exactly alike. Turbo Prolog's set of instructions and commands differs from that of any other Prolog. What's more, Turbo Prolog comes with its own *environment* for programming: a set of windows and menus to help you write, compile, run, and debug programs. This environment even includes an *editor* or simple word-processing program for writing the lines of your programs before you save and compile them. (Many pro-

programming languages used to come without an editor, figuring that you would supply your own. The popularity of Turbo Prolog and Turbo Pascal has changed this situation, so that most programming languages for microcomputers now come with an editor.)

The Turbo Prolog environment was modified in version 2.0 of the language, with some new commands added and some old commands moved to different menus. Another, larger change was the addition of *hot keys*: quick ways to get things done without using the menus. The new environment is also very flexible, offering you lots of leeway in customizing the key commands and help lines to fit your own desires and situations.

This chapter briefs you on the menus and windows of Turbo Prolog. Chapter 3 describes how to use the built-in editor, a requirement before you write and modify programs. (If you want to move on to actual Prolog programming, skip to the beginning of Chapter 4.) The menus and windows are like those in other Borland languages, the editor uses the same basic commands as the Notepad in SideKick (or as the WordStar word processor from MicroPro), and you can get help information while you work by pressing the F1 key.

Starting the Program

As described in Chapter 1, Turbo Prolog 2.0 comes with installation utilities that unpack and store—in the proper subdirectories—the compressed files from the original four Prolog floppy disks. Once you have set up those directories, either on your hard disk or on five floppy disks, you can begin Prolog simply by typing **prolog** at the DOS prompt and pressing ENTER. You'll see the main display with a copyright notice and version number in the middle of the screen. The menu bar will run across the top and four windows will be dispersed across the screen. An example is shown in Figure 2-1. Press ENTER or any key to eliminate the copyright notice. If a file named **WORK.PRO** is on the disk, it will automatically load into the Editor window, as you can see in the message in Figure 2-2.

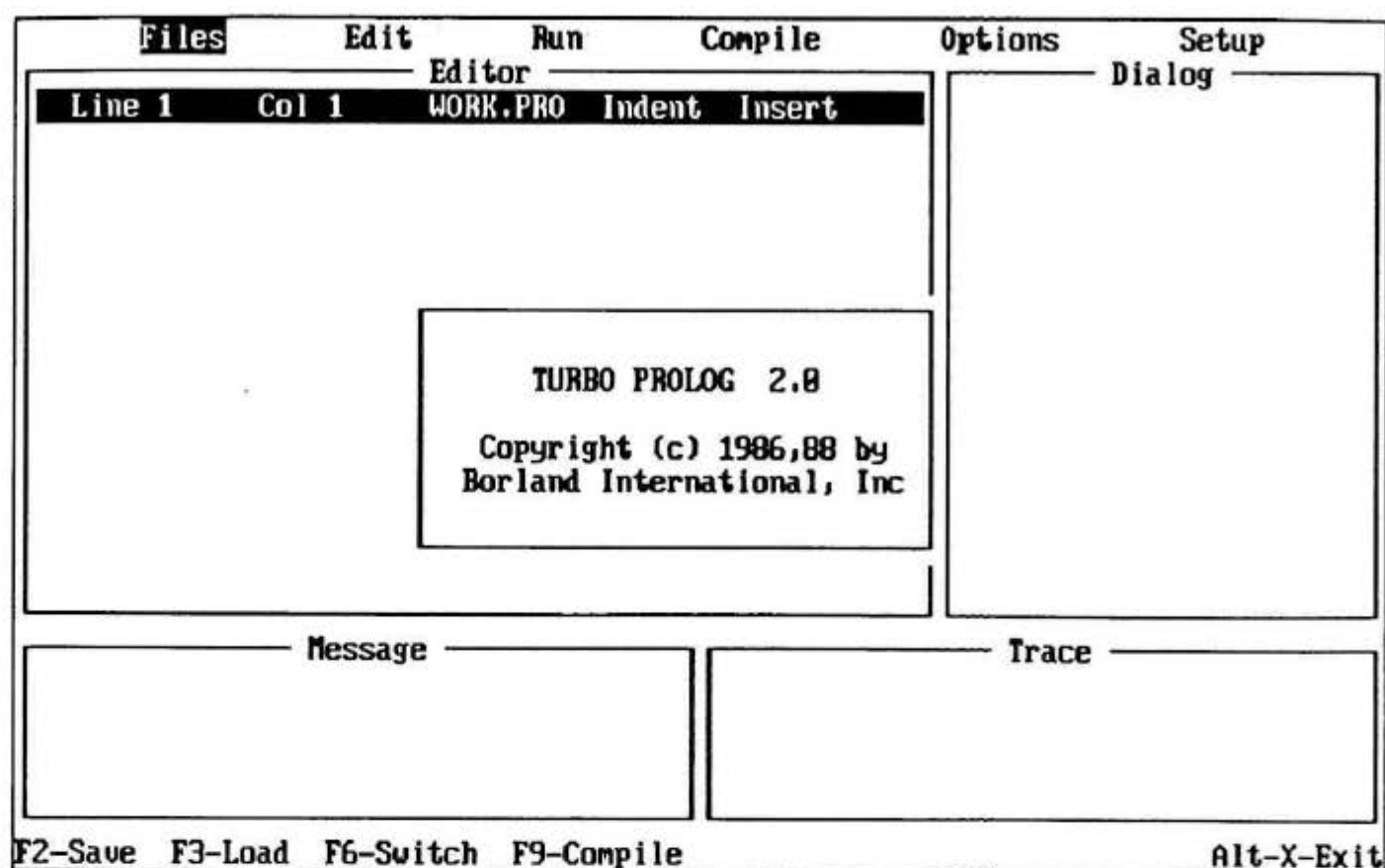


Figure 2-1. Copyright and version number screen

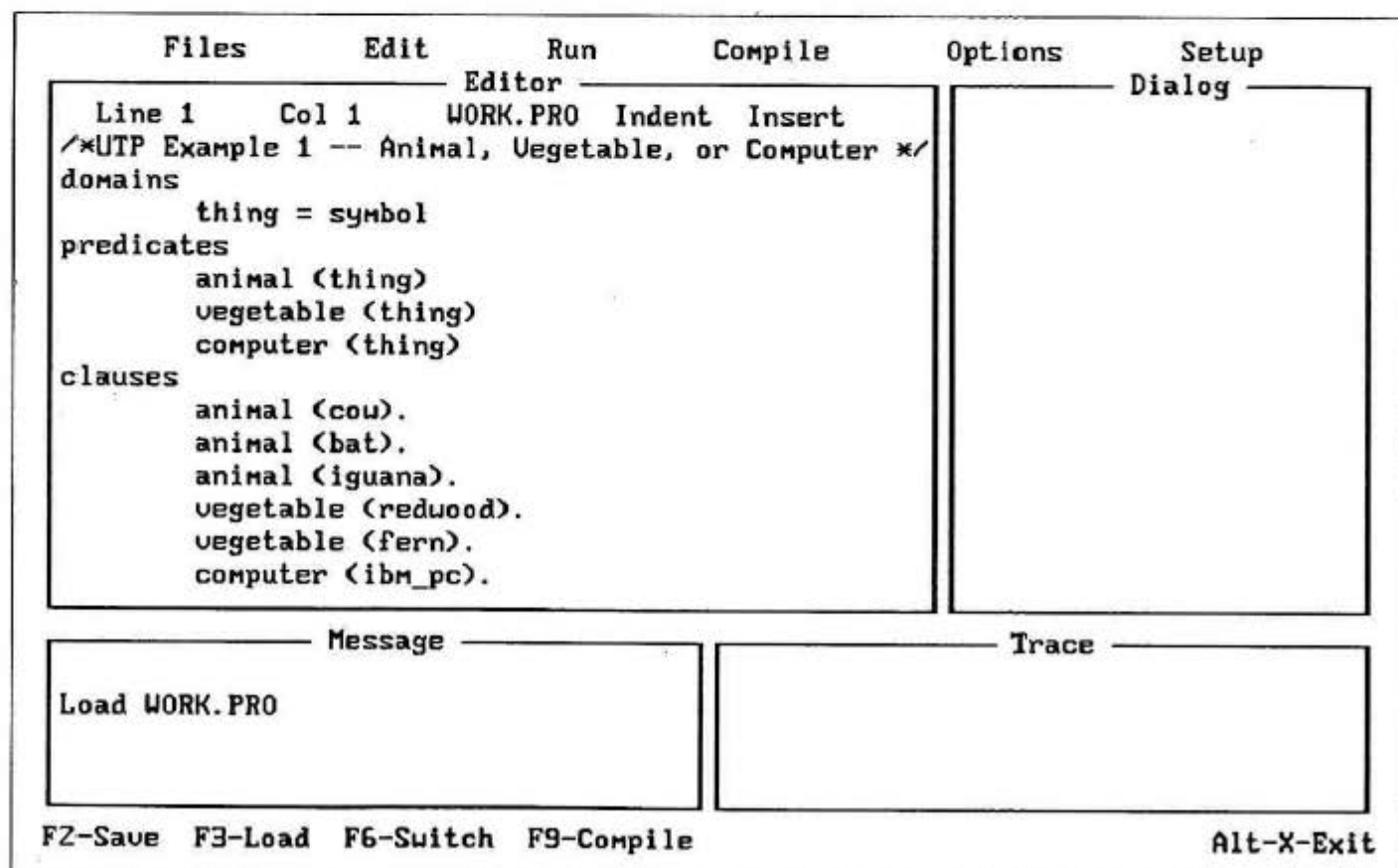


Figure 2-2. Initial Turbo Prolog 2.0 display

At the bottom of the display is an information line that lists the current commands assigned to the numbered programmable function keys. These are the the F or function keys on a personal computer keyboard. On the original PC they are the ten keys found in two columns on the left edge of the keyboard; on later keyboards they are arranged in a single line across the top of the keyboard. The operations you can perform by pressing these keys change, depending on where you are in the Prolog environment and on what part of that environment you are using. Of the initial set of definitions, Save (F2 key) refers to the file displayed in the editor area. In this case that's the WORK.PRO file, and F2 will save the displayed file to disk under the WORK.PRO name. Load (F3 key) will start the process of loading another file into the editor. The Compile command (F9 key) will attempt to compile the file in the window. The Switch command (F6 key) moves the attention of Turbo Prolog—the status of the current, active window—from one window to the next among the four on-screen windows described later in this chapter.

The last selection on that initial line, Alt-X-Exit, is actually a hot-key combination. At any point in your work you can just press the ALT and X keys at the same time (the X does not have to be capitalized, so you don't have to hold down the shift key) to exit from Prolog altogether and move to a DOS prompt. (Other hot-key definitions can be seen by pressing ALT-H, as shown in Figure 2-19.)

The four rectangular, bordered areas in the middle region of the screen—labeled Editor, Dialog, Message, and Trace—are the windows of Turbo Prolog. Each provides a function in writing, modifying, testing, and running programs. The six *menu items* across the top of the screen—Files, Edit, Run, Compile, Options, Setup—help to organize the many specific commands that you can apply to the Prolog environment, including the customization of the environment and windows.

Windows

Windows are separate portions of the computer screen that act like complete screens within themselves. In a system without windows,

switching from one task to another—for instance, switching from editing a program to running a program—often means erasing all elements of the Edit screen and replacing them with the Run screen. The windowing system provided in Turbo Prolog, allows you to display both the Edit screen and the Run screen at the same time, and switching between them is as easy as pressing a hot key.

Turbo Prolog's windows are built into the compiler and are customized to handle the four major aspects of Turbo Prolog programming: editing, querying, tracing, and debugging. Each window can be moved, reduced, or enlarged, depending on your own needs. The Turbo Prolog language can also be used to create windows for the programs you write. These windows are separate from the environment windows, but have a similar appearance and function. (The Turbo Prolog Toolbox, an optional package of Prolog routines and commands that is described briefly in the last chapters of this book, contains tools for quickly making, positioning, and labeling windows.) The windows have not changed from versions 1.0 and 1.1 of Turbo Prolog, though some of the commands for manipulating them have.

Using Turbo Prolog isn't the only way to induce your PC to "do windows." You could also buy special windowing software that adds to your operating system and creates a windowing environment to work with your other application programs. With such a program, your PC would allocate a window to each separate program you run. This can be handy if you have a language without a built-in editor or debugger, and want to have those tools close at hand as you program.

Sizing and Moving Windows

If you press the F6 key from the initial display, you'll see the border around the Trace window change to a double line. This indicates that Trace is now the active or top window. Press F6 again and the Editor window will become the active window. At this point, the windows are active only for resizing or moving.

As you can see in Figure 2-3, some of the function key definitions changed with the change of active window status. F6 kept its definition; press it again and the active status will move to the next window. Press F10 and no windows will be active. Instead, the

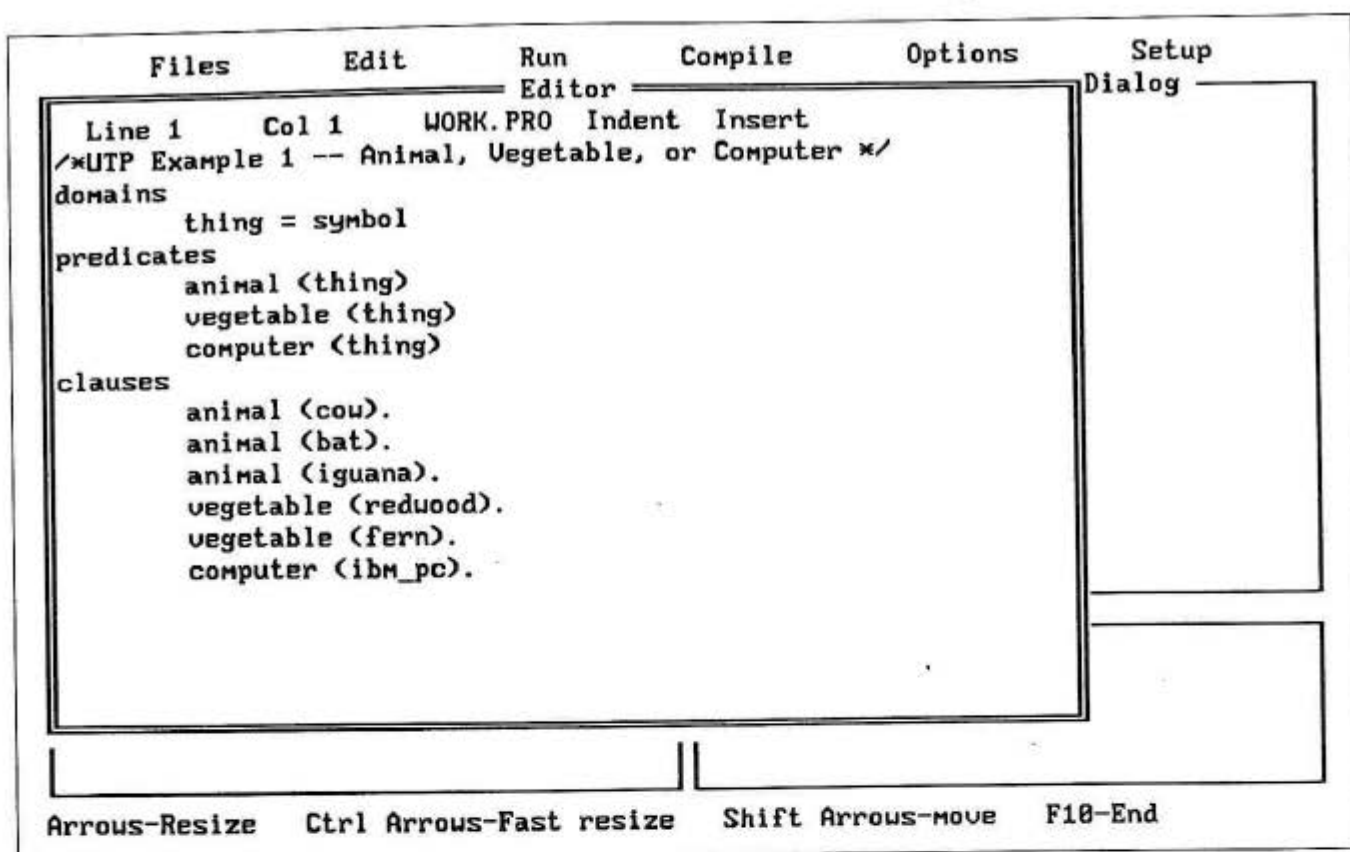


Figure 2-3. Resizing the active Editor window

menu bar will be the center of activity again. The ESC key will have this same effect.

The Arrows-resizing definition means that you can use the cursor-arrow keys—the down-, up-, left-, and right-pointing arrow keys—to change the size of the active window. Simply pressing the keys will move the righthand side and bottom border of the window in or out. Pressing the CTRL (Control) key at the same time you press one of the cursor-arrow keys will make larger moves in the same direction. Pressing a SHIFT key and a cursor-arrow key at the same time will move the lefthand side and top border of the window. While you press these keys, their definitions are shown on the information line at the bottom of the screen. Figure 2-3 shows an enlarged Editor window that overlaps the other windows.

Trace Window

Turbo Prolog includes several commands for *tracing* or closely inspecting the processing of a program. You can follow the behav-

ior of a program you have written, watching its action step by step to see if it is doing what you want it to, or to determine why it is not. The actions (or *traces*,) of your program are displayed in the Trace window. The commands for setting them in motion are explained throughout this book. If you're not tracing a program, you can reduce this window to an insignificant size to make more room for the other windows.

Message Window

This is the window the Prolog environment uses to keep you informed. If you try to save or load a program, that fact will be noted in the Message window (see Figure 2-2 for an example). If you tell Turbo to compile a program, that will be noted in the window, and you can even see the various predicates or commands of the program compile as Prolog shows you its progress. You can often afford to make this window only a few lines tall, and fairly short. Sometimes, however, it is important to see what the system has tried to do.

Dialog Window

As you'll see when working through the examples of this book, using simple, logical Prolog programs often consists of asking a series of questions of the program—using Prolog syntax, of course. The Dialog window is where this takes place. When you run many simple programs, they will offer you the word

Goal:

in the Dialog window. This means that the program is waiting for you to type in some goal that the Prolog logic can test. The Dialog window is a real necessity in simple programs, and you'll want to afford it some space, especially in width (so that lines don't wrap confusingly from one to the next). In sophisticated programs, you'll probably use input and output features that bypass the Dialog window, and you can then squash it down to make more room for the debugging abilities of the Trace window and the long lines of code in the Editor window.

Editor Window

This is the window you'll use most frequently. Even in the initial configuration of Prolog this is the largest window, because this is where you view and work on your programs. A Turbo Prolog program is a list of instructions written as lines of text and numbers. The Editor window not only gives you a place to inspect these lines of *source code*, but also has a built-in set of commands for moving and editing the lines. These commands are called an *editor*, and make up a straightforward, simple word or text processor. The commands follow much of the WordStar standard, named after a best-selling word processor. The same basic commands were included in Borland's SideKick Notepad utility.

You could even use this editor (described in Chapter 3) to write and save memos, papers, or any other sort of text document, though this might not be efficient; it wasn't designed to format and print those kinds of files with any aplomb. Within the Editor window, incidentally, the F1 key is the Help key. The top line of the Editor window tells you the line and column where the editor's cursor is sitting, the name of the file in the window, and whether Indent and Insert modes are on or off. This is all explained in more detail in Chapter 3.

The Auxiliary Editor window has the same editing commands as the main Editor window and let's you edit another file without losing the file in the main editor. The Auxiliary Editor is explained further in Chapter 3.

Saving a Window Configuration

After you move and resize the windows to your particular liking, you can save their position and size in a configuration file. In fact, you can save a number of configuration files under different names, and call them back to quickly change the environment's look whenever you want to. You might have an editing configuration and a tracing configuration ready to go at a moment's notice. To do this, use the Setup menu's Save configuration command, explained under the Setup menu section later on in this chapter.

Menu Bar

The line of menu titles across the top of the screen is the *menu bar*. Four of these titles — Files, Compile, Options, and Setup — contain a variety of related commands beneath them in a *pull-down menu*. Some of these menus even have other menus tucked into their selections, making menus within menus for specialized commands. Two of the menu bar titles — Edit and Run — are single commands, with no pull-down specifics beneath. The menus have been changed in many ways from versions 1.0 and 1.1 of Turbo Prolog, though many of the old commands have been retained.

When the menu bar is active, you can select one of its titles in three ways. One way is to press the left and right cursor-arrow keys to move the highlighting to that title, then press the ENTER key. Another way is to press the key for the first letter of the title. The third way is to use the *hot key*, or shortcut key combination, designated for that menu. If you choose Edit or Run, those commands will be executed immediately because there are no more choices to make. If you choose one of the other four titles, you'll see a menu "pull down" or move downward from that title, a menu containing more specific choices that must be made before Prolog can initiate any action. Figure 2-4 shows the Files menu beneath the Files title in the menu bar.

You select menu items in the same way you choose menu bar titles: either by using the cursor-arrow keys and the ENTER key (the up and down keys in this case) or by pressing the first letter of the menu option.

Borland divides the menu choices into three types: commands, toggles, and settings. *Commands* perform a task, *toggles* turn a feature on or off, and *settings* specify a numeric value or name for a feature. You don't really need to know which is which ahead of time; it will become clear as you see and use the different choices.

The ESC Key

An important key is ESC. It is assigned a single function that remains constant no matter where you are in Turbo Prolog.

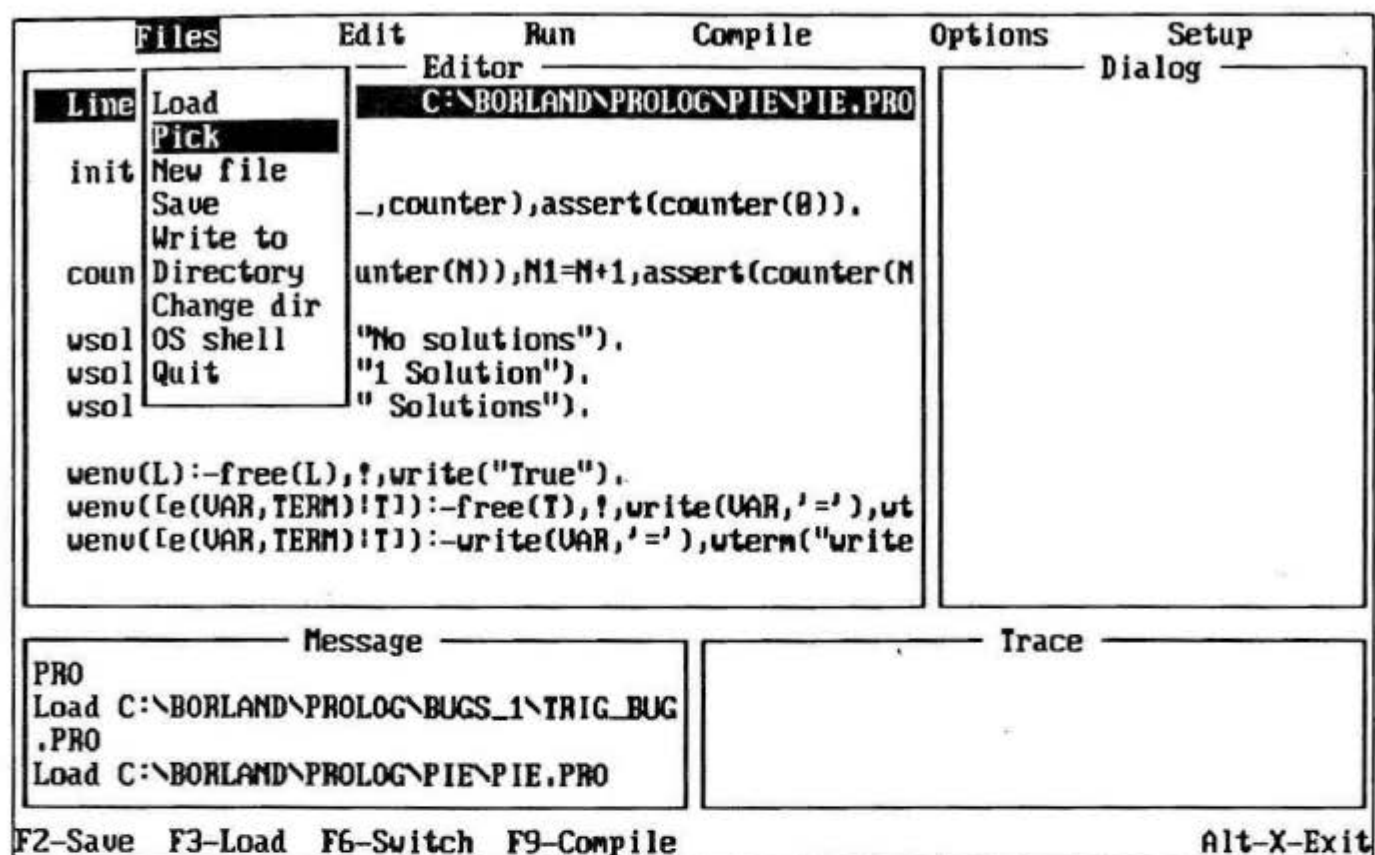


Figure 2-4. Files menu

The ESC or Escape key is most often found at the top left or right corner of a personal computer keyboard. It “escapes” you from your present situation in the Prolog environment. That is, if you are in a window and want to get back out to the main menu, press ESC. If you are several levels down in the pull-down menus, each time you press ESC you’ll move back up a level. If you’ve typed some information you’re not sure you want to keep, just press ESC to jump out of the question area and back to a previous menu. (All of these choices will be explained in more detail later.)

Files Menu

The Files menu (see Figure 2-4) contains nine commands. They differ from the commands of versions 1.0 and 1.1 of Turbo Prolog, but are suited to the same tasks: manipulating disk files and Editor window files. Load is for moving a copy of a disk file into the Editor window. Choose this command and you’ll see a new, smaller

window asking you for the name of the file you want to load (see Figure 2-5). This window presumes that the file extension will be PRO.

If you press ENTER without typing a full file name, or after typing a name with some wildcard elements, you'll then see a Directory window of the current disk directory and subdirectory, as shown in Figure 2-5. (*Wildcards* are symbols that can stand for any letter or number in a file name, and so help you select files for which you aren't sure of the exact name. Prolog uses standard DOS wildcards: a question mark for any single unknown character and an asterisk for any number of unknown characters in a row.) Names shown in the Directory window that have a slash after them are subdirectories, not individual files.

To select one of the files in a directory, you can use either of the schemes you use with a menu: Press the first letter of the file name or move the highlighting to the file name. Then, in either case, press ENTER. If you move the highlighting to a subdirectory and press ENTER, the Directory window will change to show any files within that subdirectory. If you choose "... \", you will be moved one subdirectory level up.

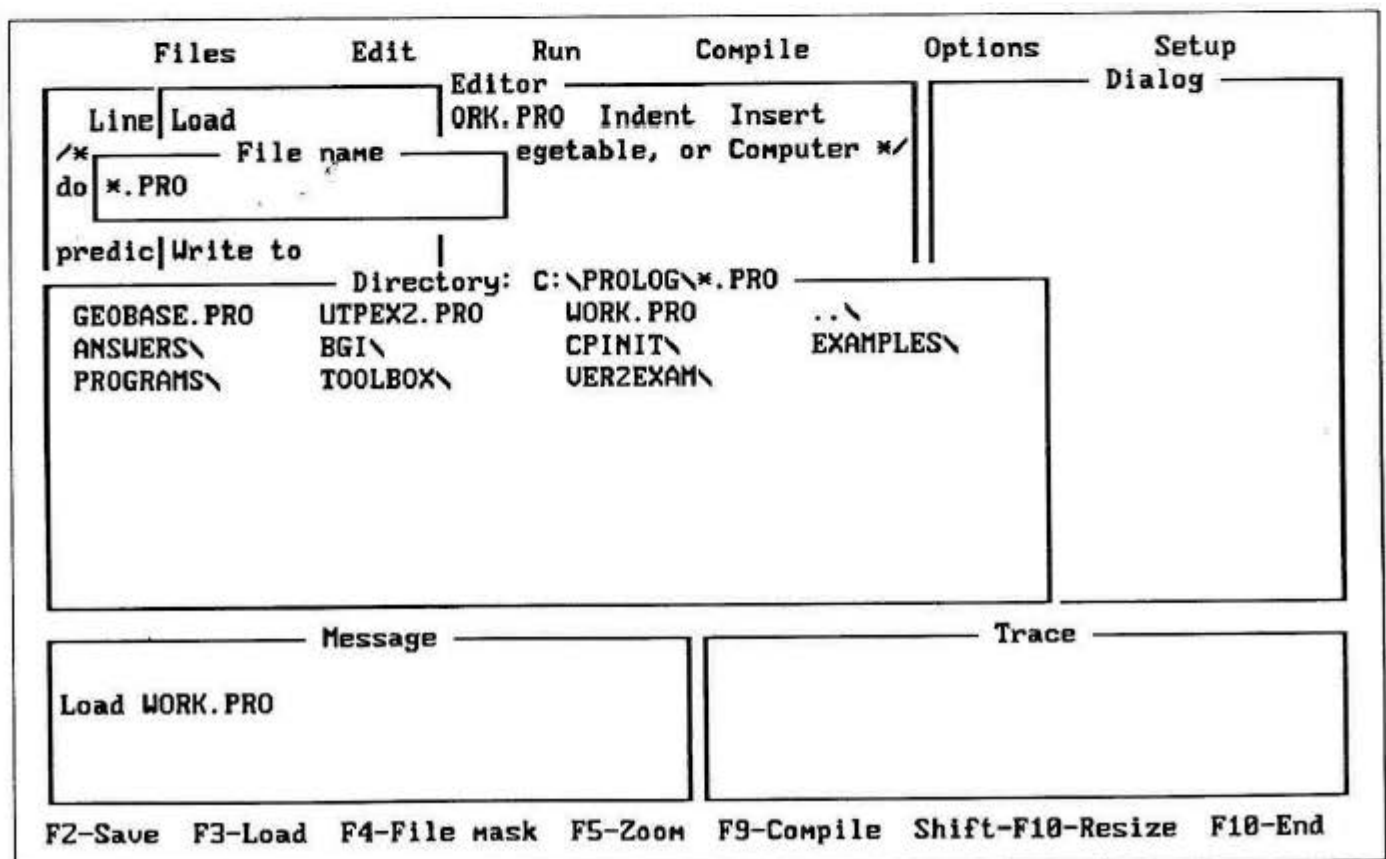


Figure 2-5. A Directory window

If Prolog can find and load the chosen file, that file will appear in the Editor window; the Files menu will disappear and you will be placed in the editor. The Message window will bear witness to what just happened. If you want to extricate yourself from a Directory window or some other part of a menu choice, press ESC as many times as necessary to move back to a comfortable menu position. Pressing ESC skips making and executing the actual, highlighted, menu choice.

Directories “Directory” and “Change dir” let you set the *mask*, or wildcard selection criteria, and the disk drive and directory that will be employed for other File menu choices.

Saving Files “Save” saves the current file in the Editor window to disk using the name at the top of that window. If there is already a file on disk with that name, the old file will be copied to a backup file before the new file is saved on top of the old file. “Write to” lets you save the file in the Editor under a name given at the time of the save. “New file” lets you clear the Editor window and start a new file using the name WORK.PRO as the file name. You can also save files with hot keys or Editor window commands, as described later in this chapter and in Chapter 3.

DOS Access “Os shell” lets you pop out to DOS without totally removing Prolog from memory. Figure 2-6 shows how you’ll jump to a blank screen with the DOS copyright notice. You can then execute DOS functions by typing their standard names (such as DIR, CD, or whatever). When you’re done with DOS, type **exit** and press ENTER to get back to Prolog. This DOS access takes less time than leaving Prolog and loading and starting it again, and lets you move files and directories more easily than with Prolog’s limited File commands. “Quit” naturally leaves Prolog altogether, much as the hot key ALT-X does.

Pick An addition to the Prolog 2.0 command repertoire is “Pick,” which remembers the last eight files you loaded and lets you choose to load any one of them again without working through directories and masks. Figure 2-7 shows how the Pick choice gives you a list of those files, with the most recently loaded file at the top. If you want to throw away changes to the file in the Editor window, and haven’t

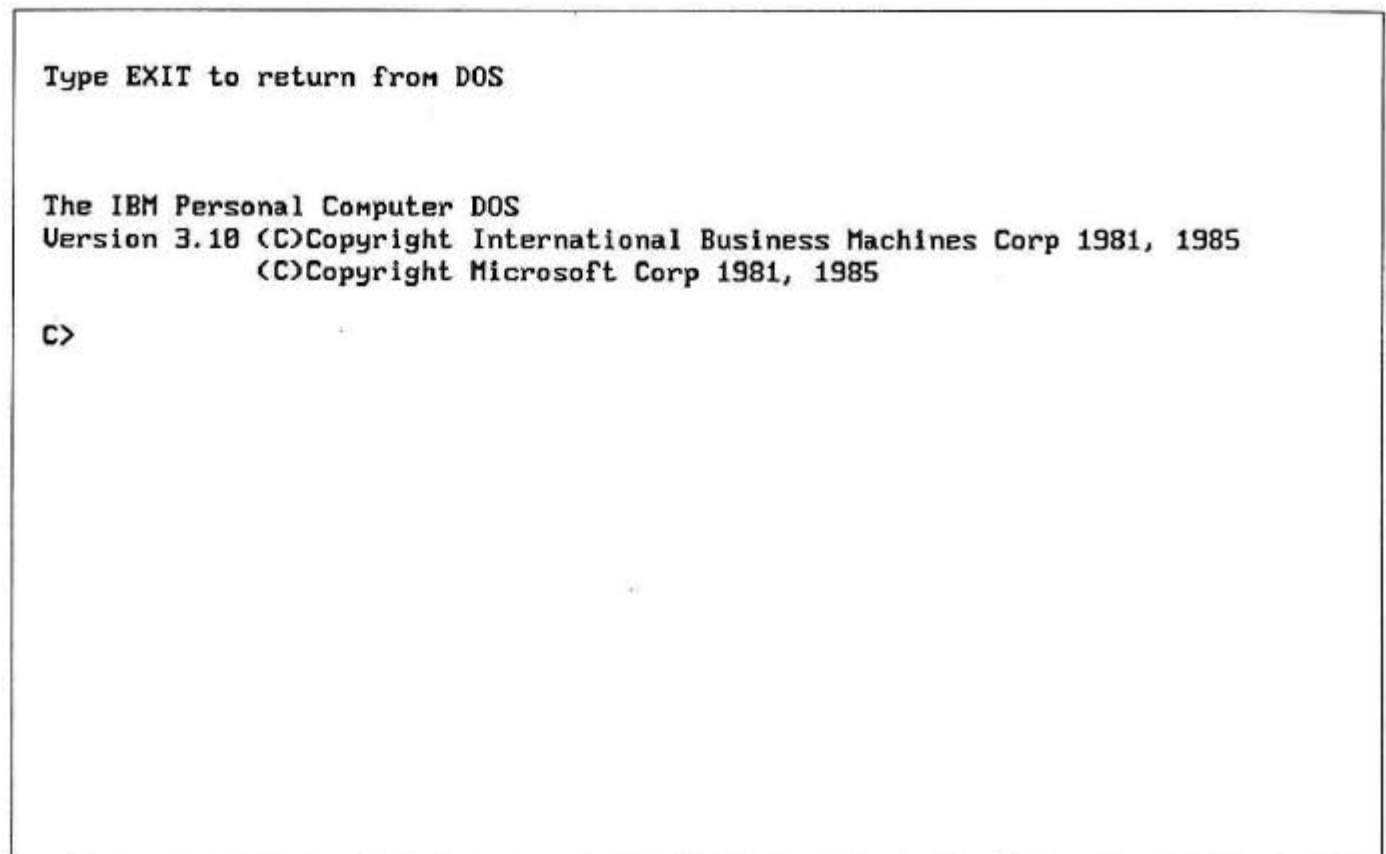


Figure 2-6. DOS access screen

saved those changes to the disk version of the file yet, just choose the top file on the Pick list to load again. The top menu item (Load file) allows you to enter a new file name (or filemask) that you want to load.

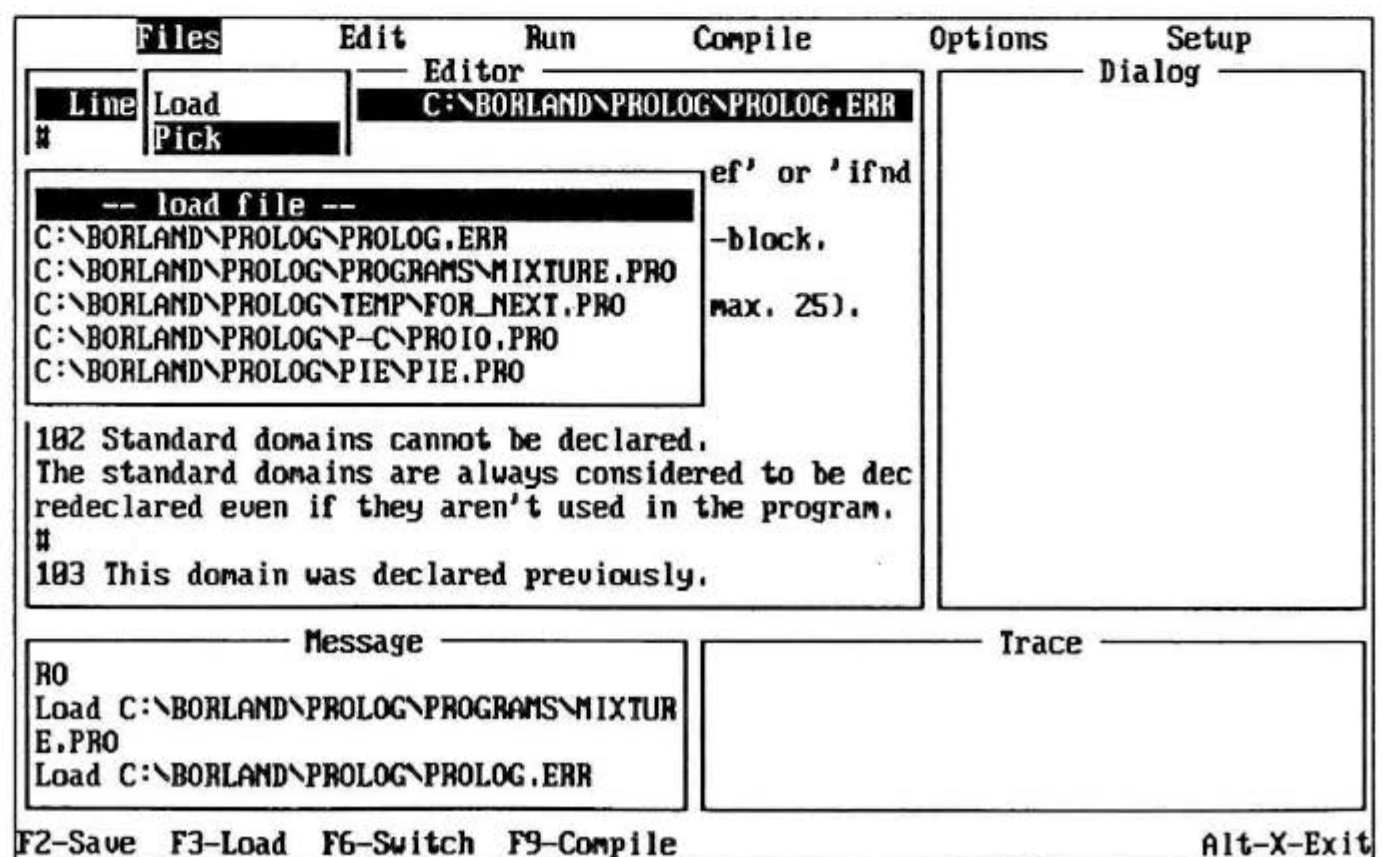


Figure 2-7. Pick directory for reloading recently loaded files

Edit Menu

This isn't really a menu at all because it is only a single choice. Choose "Edit" and the Editor window, along with its built-in editor, will be active, shown with a double-line border. You can then use this window and its commands to modify or create Prolog source-code files, as explained in Chapter 3. Any typing you do will show up in the Editor window and won't relate to choosing menu options. To return to the main menu use the ESC or F10 key.

Run Menu

This is no more a menu than is the Edit title, again because there are no choices tucked underneath the main item. Choose "Run" and Prolog will attempt to compile and run the file in the Editor window. If the file in the Editor window has been changed since it was last run, the Run choice will attempt to compile and then to run that program; otherwise the program will run without being recompiled.

Compile Menu

The Compile menu combines some old commands that were scattered throughout Prolog 1.0 and 1.1 menus with some new commands. When you *compile* a program, you are telling Prolog to transform your source-code file of English-like phrases and commands into the machine language ones and zeros that a computer actually runs on.

There are five choices on the Compile pull-down menu, as shown in Figure 2-8. "Memory", "OBJ file", and "EXE file (auto link)" will each attempt to compile the current Editor window file into memory, as a disk-based OBJ file or as a disk-based EXE file, respectively. (These types of files will be explained later. Simple and practice programs are normally compiled to Memory.) The "EXE" option automatically links together the various components

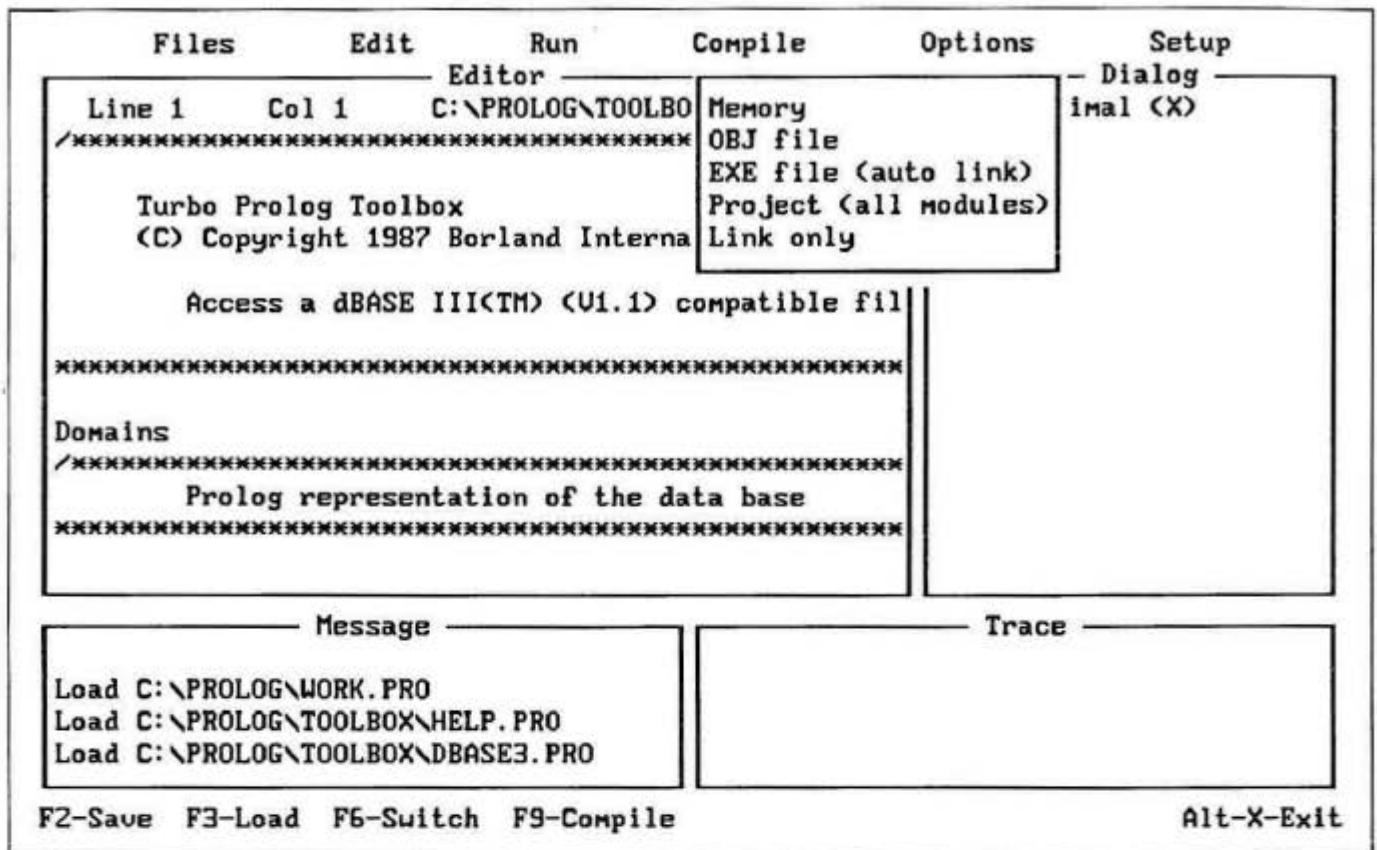


Figure 2-8. Compile menu

of a complex compilation. Some compilers have separate linker utilities that you have to use to join pieces of a large program together. "Link only" is for linking together different pieces of Prolog code that are already compiled. "Project (all modules)" is for compiling a program from a group of Prolog modules or files. Projects and Linking are explained in detail later in this book.

If you compile a program to memory, you can then immediately run it from inside the Turbo Prolog environment. (The Run option, as mentioned above, compiles to memory and runs a program in one step.) If you compile to an OBJ file (which will have the letters OBJ as its file-name extension), you can use the object code later in sophisticated projects and programs. Compiling to an EXE file lets you make a program that can then be run without the use of Turbo Prolog, a program file that you can copy to other disks and run on other computers that don't have the main Prolog files. The EXE file contains all it needs from those files to run independently, along with your own instructions and commands written in the Editor window.

Options Menu

The Options menu contains more advanced compilation choices, most of which are explained in more depth in the advanced chapters of this book. The "Link options" and "Edit project definition" choices let you specify how the compile options will run. The "Compiler directives" option brings up a submenu with a set of compilation aspects that you can toggle on or off or set to a value.

Even within this submenu there are still more specific menus, as shown in Figure 2-9. Here you see the three Trace options that can be selected. The toggle choices change from on to off or from off to on each time you select them. The other choices, including Trace, come with submenus for selecting specific values. If you want to use some other setting than the default values, you set these directives and toggles before using the Compile menu.

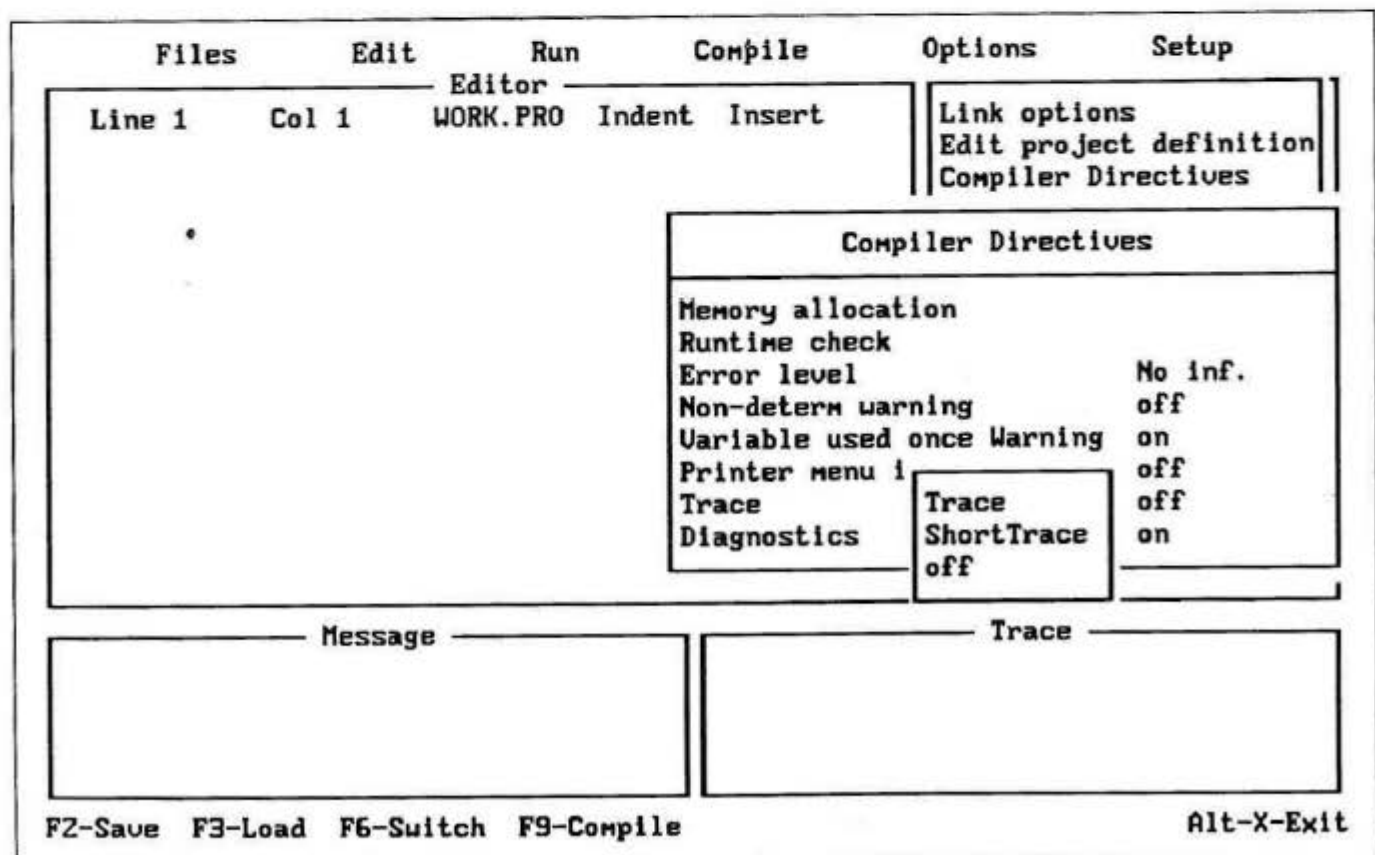


Figure 2-9. Options menu and Compiler directive submenu

Setup Menu

This is the housekeeping menu. It lets you inspect, temporarily modify, or permanently change the configuration of your Turbo Prolog environment. Figure 2-10 shows the Setup menu along with the "Save configuration" file-name window.

The *configuration* of Turbo Prolog is the set of specifications for window size and position, colors, directories for finding and saving files, screen display mode, keyboard configuration, and so forth. This configuration is saved as a file with a .SYS file-name extension. The default configuration file is PROLOG.SYS; this is the file that Prolog will try to load each time it starts. You can save separate configuration files with various names, calling them up later (using the Load configuration option) and instantly changing all of the configuration values for the environment. If you save a new .SYS file as PROLOG.SYS, that is the configuration that will load when you first bring up Turbo Prolog.

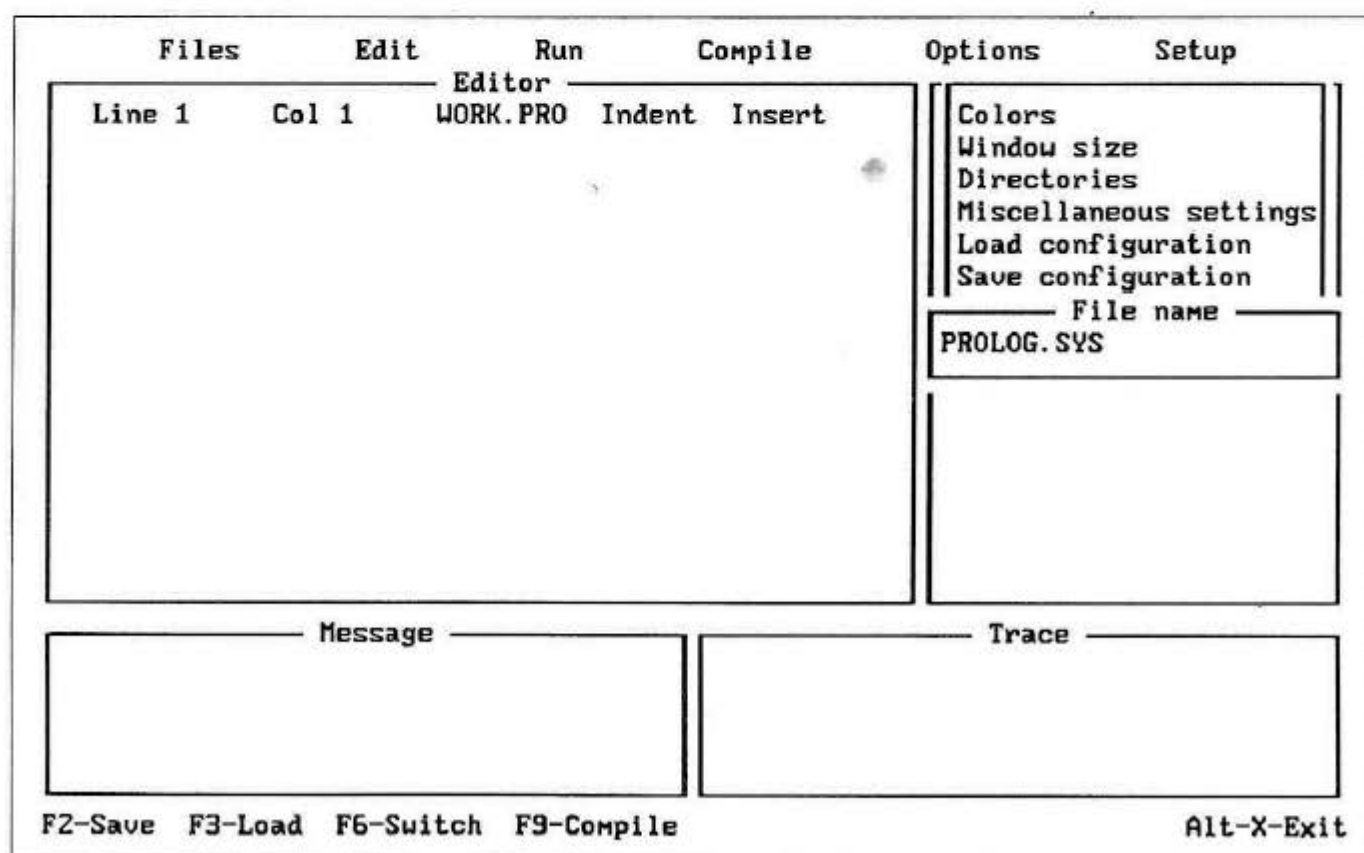


Figure 2-10. Setup menu and Save configuration option

Colors The Colors submenu, shown in Figure 2-11, lets you change the colors for each of the windows or screen elements: Editor, Dialog, Message, and Trace windows, as well as menus, status line, and Auxiliary Editor window. The box that is left-center of the screen in Figure 2-11 shows how this works (though in black and white it is not so convincing). You can move the color selection marker around within the possible colors, seeing the number of the color on the top line and the actual color painting the copyright notice. For windows, you can select both the interior and the border color. Even if you have a monochrome display, some color combinations will show more clearly than others. You no longer have to bother with the color attribute values from versions 1.0 and 1.1 of the program (though you can use these numbers later on when you choose the color attributes to use in your programs).

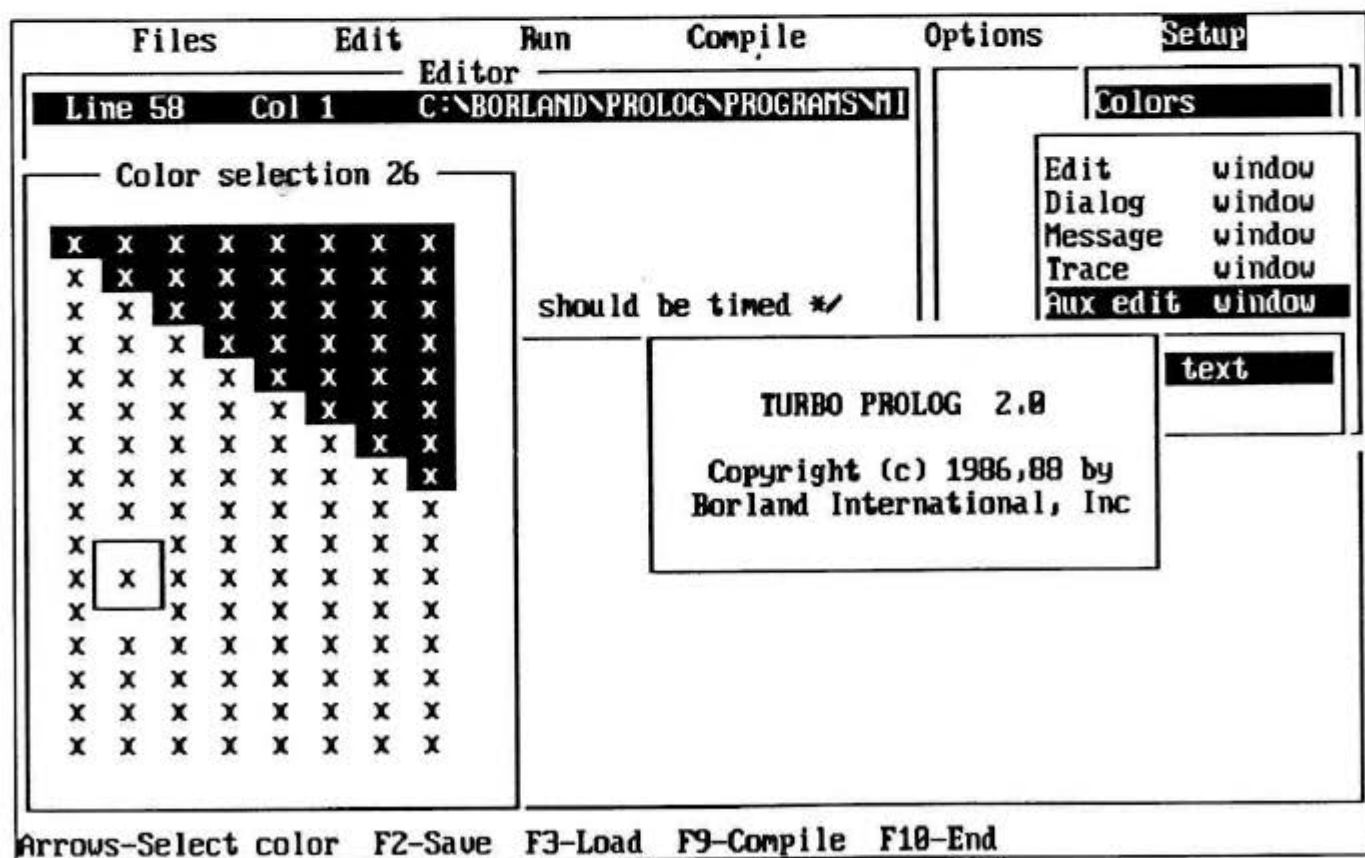


Figure 2-11. Colors submenu with color selection box

Window Size You can change the size and position of windows without the Window size menu, but the menu does make it a bit easier by organizing the task. As mentioned previously, you use the arrow keys, in concert with the CTRL and SHIFT keys, to move and position the borders of the windows. You can also use these keys:

HOME	To extend a window to the left edge
END	To extend it to the right
SHIFT-HOME	To move it to the left edge
SHIFT-END	To move it to the right edge
CTRL-HOME	To move it to the upper lefthand corner
CTRL-END	To move it to the bottom righthand corner

As an example, adjusting the Auxiliary Editor window is shown in Figure 2-12.

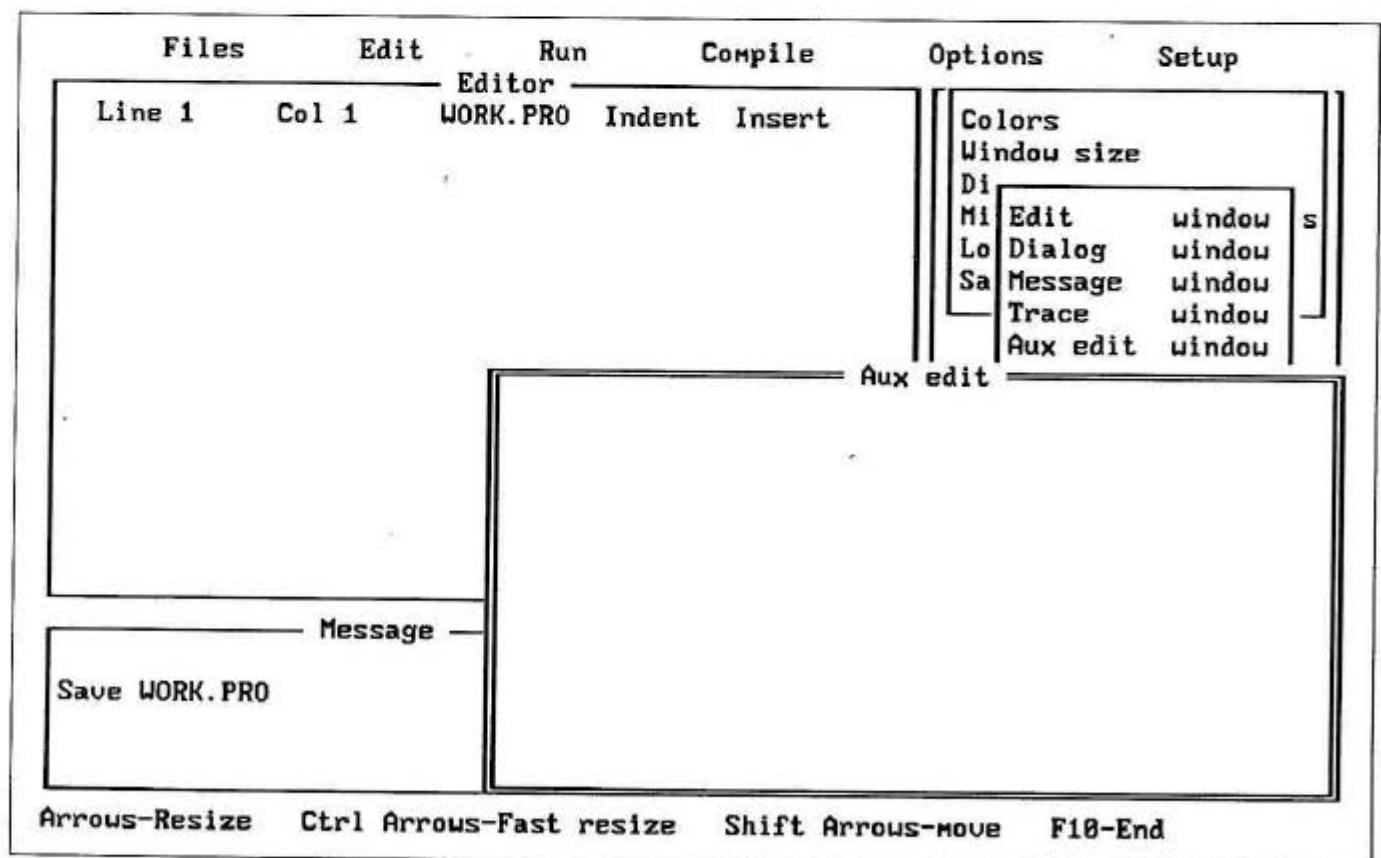


Figure 2-12. Window size menu with bottom-line instructions

Directories The directories you specify from the Directories submenu, as shown in Figure 2-13, tell Prolog where to find and save various kinds of files. The PRO directory is used for all file-handling operations in the Files menu. (You can also change that directory within the Files menu itself.) The OBJ directory is for files with the OBJ file-name extension. The EXE directory is for EXE files compiled by Turbo Prolog, and the Turbo directory is for the Turbo Prolog system files—the files of the compiler, such as error messages and help messages. If you leave a line blank and press ENTER, you'll get a Directory window, just as you would in the Files menu. This allows you to "walk" to the directory you wish to select. Pressing ESC brings this directory into the Directory choice menu, and pressing ENTER selects this directory.

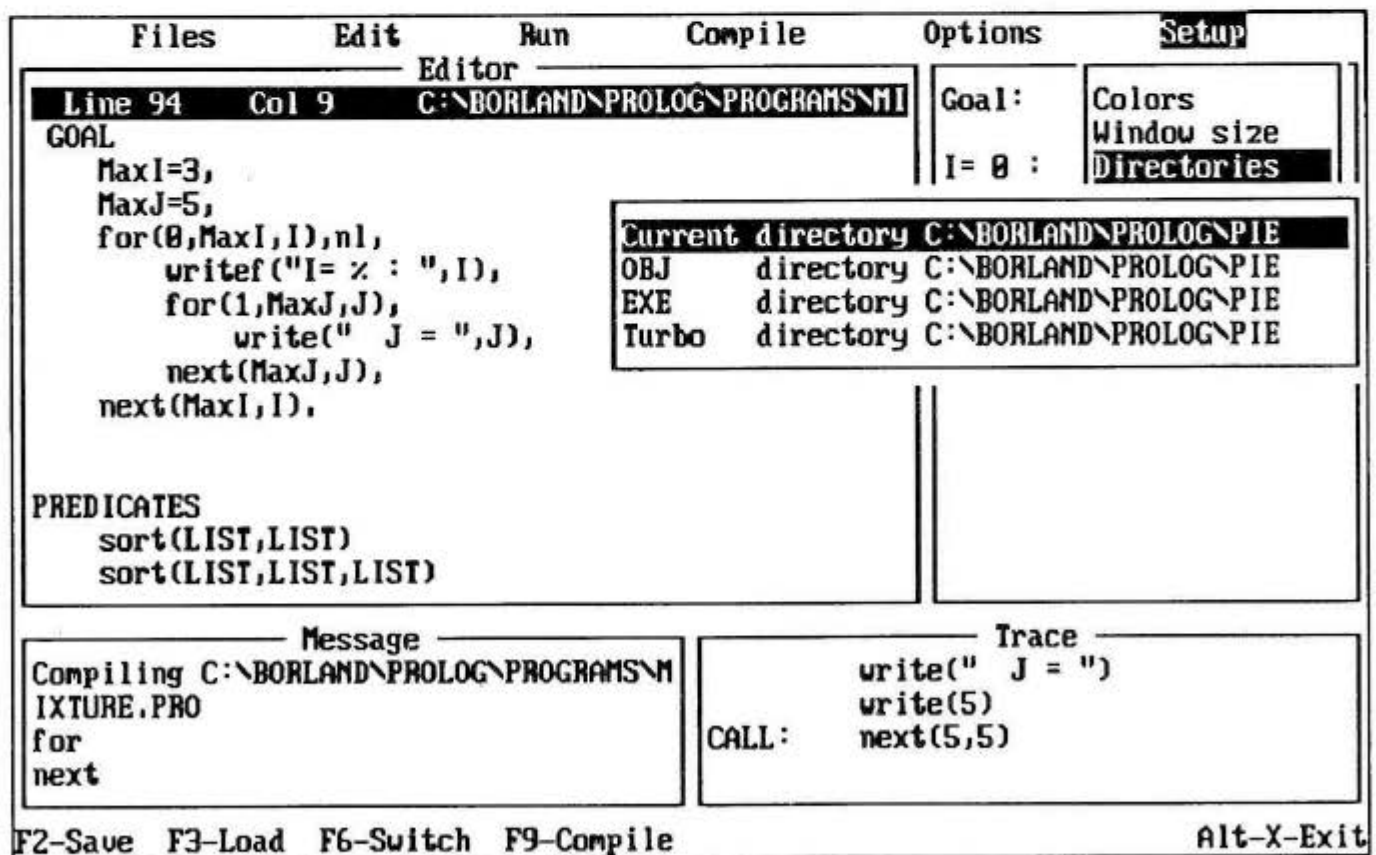


Figure 2-13. Directories submenu

Miscellaneous Settings These are what are left over after the neat organization of the other menus and submenus, and are shown in Figure 2-14. Each time you press **I** or **A** you toggle the CGA adapter setting or the Auto-load message on or off. If the IBM-CGA adapter choice is on, special synchronization will be employed to generate a better color screen display for CGA cards. If the Auto-load message is turned on, the error messages of Turbo Prolog will be loaded into memory when you load the compiler. Normally this choice is off, to save memory space, so each time an error appears in compiling, the relevant error message has to be found and loaded from the disk. The drawback to saving memory is that reading the disk can slow down operations.

Files	Edit	Run	Compile	Options	Setup
Editor Line 1 Col 1 WORK.PRO Indent Insert				Colors Window size Directories Miscellaneous settings	
				IBM-CGA Adapter off Auto load message off Screen mode 25x80 Keyboard configuration Help lines	
Message Load WORK.PRO			Trace		
F2-Save F3-Load F6-Switch F9-Compile Alt-X-Exit					

Figure 2-14. Miscellaneous configuration settings

Screen Mode Choose "Screen mode" and you'll see the menu in Figure 2-15, where you can select from a variety of screen display modes that will put different numbers of rows and columns of text on your display. You can only use modes that are available on the type of adapter you have. The EGA 43-line by 80-column mode can be found in a number of software programs, and lets you see more text on the screen at once, though in smaller, harder-to-read characters. Care should be taken here to choose a mode that your adapter supports.

Keyboard Configuration If you don't like the keys you have to use to execute commands in Prolog, version 2.0 comes with this option to customize your own set of key equivalents for operations. The menu shown in Figure 2-16 lets you choose the type of commands you want to modify, the individual commands to change, and then the keys you want to use for those commands.

Figure 2-17 shows how an individual command can be selected from a list (in this case, the WordStar block commands). You are given a blank slot to fill with whatever key or key combination you

Files		Edit		Run		Compile		Options		Setup																															
Line 1	Col 1	Editor		WORK.PRO	Indent	Insert		Colors Window size Directories Miscellaneous settings																																	
								IBM-CGA Adapter off																																	
								<table border="1"> <thead> <tr> <th>Rows</th> <th>Columns</th> <th>Requirement</th> </tr> </thead> <tbody> <tr><td>25</td><td>80</td><td></td></tr> <tr><td>25</td><td>90</td><td>MultiScan</td></tr> <tr><td>25</td><td>120</td><td>EGA+</td></tr> <tr><td>25</td><td>132</td><td>MultiScan</td></tr> <tr><td>43</td><td>80</td><td>EGA</td></tr> <tr><td>43</td><td>90</td><td>MultiScan</td></tr> <tr><td>43</td><td>120</td><td>EGA+</td></tr> <tr><td>43</td><td>132</td><td>MultiScan</td></tr> <tr><td>50</td><td>80</td><td>UGA</td></tr> </tbody> </table>				Rows	Columns	Requirement	25	80		25	90	MultiScan	25	120	EGA+	25	132	MultiScan	43	80	EGA	43	90	MultiScan	43	120	EGA+	43	132	MultiScan	50	80	UGA
								Rows	Columns	Requirement																															
								25	80																																
								25	90	MultiScan																															
								25	120	EGA+																															
								25	132	MultiScan																															
								43	80	EGA																															
								43	90	MultiScan																															
								43	120	EGA+																															
43	132	MultiScan																																							
50	80	UGA																																							
Message																																									
F2-Save F3-Load F6-Switch F9-Compile								Alt-X-Exit																																	

Figure 2-15. Rows and Columns selection menu for display configuration

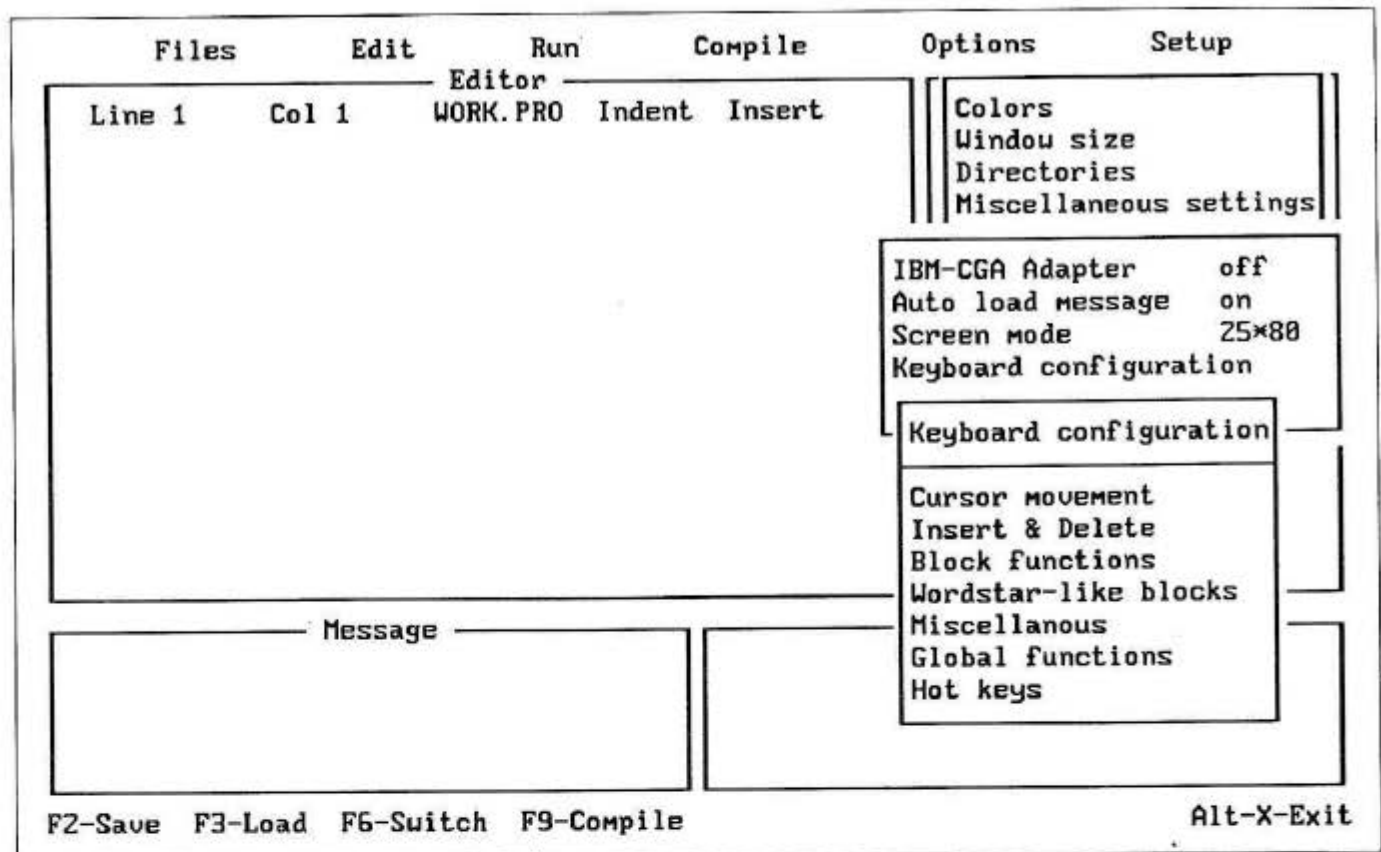


Figure 2-16. Keyboard configuration menu: command types list

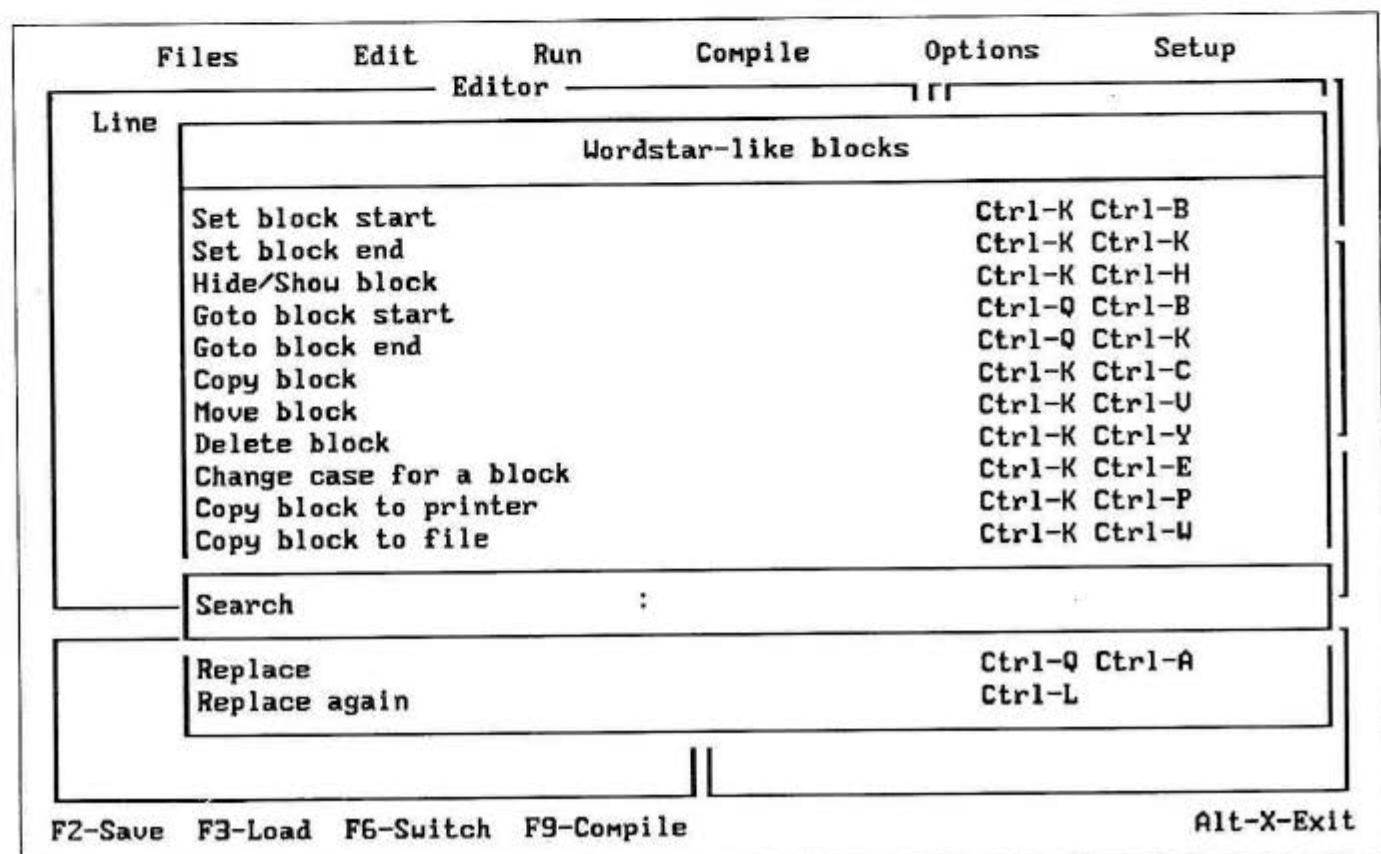


Figure 2-17. Keyboard configuration menu: changing an individual command

want to use. Your new key command will be in place, substituted for the old. Using this tool, you can make the editor resemble your favorite word processor. Working through the commands the first time may be tedious, but you can then save the new key equivalents in a SYS file and call them up anytime you want to.

Help Lines *Help lines* are the key definitions that are displayed on the bottom line of the screen. You can customize these to fit your new commands and to better help you use Turbo Prolog. Just choose a particular line, as shown in Figure 2-18, and type your new help "line." The new information will appear on the bottom line of the display and you can then save it as a configuration file. Press ESC if you don't like what you've typed, and the old help line will reappear.

Files		Edit	Run	Compile	Options	Setup
Line 1	Col 1	Help lines			ert	Colors Window size Directories Miscellaneous settings
		Main status Edit status Display status Resize status View system windows Printer log Select file name Browse directory Goal status Trace status Alter trace status Color status Error EXplanation				IBM-CGA Adapter off Auto load message on Screen mode 25*80 Keyboard configuration Help lines
Message				Trace		
Arrows-Select color F2-Save F3-Load F9-Compile F10-End						

Figure 2-18. *Help-line menu for rewriting the help information on the bottom line*

Files	Edit	Run	Compile	Options	Setup
Line	Hot keys				
	Quit Prolog	Alt-X			
	Activate Files pulldown	Alt-F			
	Activate operating system	Alt-D			
	Activate Editor	Alt-E			
	Activate Options pulldown	Alt-O			
	Activate Compile pulldown	Alt-C			
	Activate Setup pulldown	Alt-S			
	Display Hot keys	Alt-H			
	Display version info	Alt-F10			
	Load file	F3			
	Save file	F2			
	Run program	Alt-R			
	Compile to memory	F9			
	Compile to OBJ	Shift-F9			
	Compile to EXE	Ctrl-F9			
	Compile Project	Alt-F9			
F2-Save		F3-Load	F6-Switch	F9-Compile	Alt-X-Exit

Figure 2-19. Main menu hot keys

Hot Keys

An innovation in version 2.0 of Turbo Prolog is the hot key, a key combination that can make a shortcut to a particular operation. Some hot keys work from anywhere in Turbo; others are specific to the editor, to Trace mode, or to compiling. The editor hot keys are discussed in Chapter 3, the Trace mode hot keys in Chapter 4. Figure 2-19 shows the main menu hot keys, the hot keys that work from anywhere in Prolog. If you memorize these, you'll be able to compile, get in and out of the editor, save and load files, and change windows, all without using a single menu.

3 *The Editor*

A Prolog program is a series of characters that are grouped into words and phrases and stored in a file called the program *source file*. Writing, erasing, or modifying the characters in that file is known as editing the program. Most computers don't automatically know how to edit text. They may have built-in commands or operating system commands for moving a cursor back and forth on a single line, or perhaps even for deleting single letters at a time. However, special editing software is needed to allow computers to accept, display, move, and save characters with any sophistication.

Powerful editing programs are called *word processors* and have the functions and commands to format and manipulate text in many different ways. An *editor* is basically a simple version of a word-processing program. It will let you enter, move, delete, and insert characters. Many early microcomputer language compilers did not include an editor. Those that did often had an editor with few functions and obscure commands. The language designers assumed that if you had a lot of programming to do, you would use your own editor or word processor to create the source file. Once the source was complete and saved on disk, you would start the compiler, give it the name of your source file, and wait for the *object file* — the compiled program — to appear.

Turbo Prolog has its own built-in editor that contains most of the word-processing functions you'll want for creating or working on a Turbo Prolog source file. (You may also use other editors or word processors, as long as they produce pure ASCII text files. On the other hand, you can use the Turbo editor for your simple correspondence and notes. It creates ASCII files that can be used in other word processors; it does not have to produce files that directly relate to Prolog.) To use the built-in editor efficiently, you need to memorize a few key combinations that issue the editing commands. That is, you must know which keys to press to cause specific editing operations to take place.

Version 2.0 of Turbo Prolog lets you redefine those key combinations to fit whatever word processor or editor you are familiar with (as explained near the end of Chapter 2).

This chapter tells you how to use the Turbo Prolog editor's default commands. If you have experience in using other programs with similar editing commands (WordStar, MultiMate, Turbo Pascal, or SideKick, for example), you'll feel right at home with the Turbo Prolog editor, though you'll still have some extra commands to learn. The basic cursor movement commands are unchanged from versions 1.0 and 1.1 of the language, but many other commands have been altered.

Selecting the Editor

As you learned in Chapter 2, when you start Turbo Prolog the editor is not active. Instead, the main menu bar at the top of the screen is the active element. To get into the editor you need to choose the editor title on that menu bar, either by pressing **E** or by moving the highlight to the word "Edit" and then pressing **ENTER**. The Editor window, as shown in Figure 3-1, will become active with a highlighted label and a double-line border. The hot-key combination **ALT-E** will also move you into the editor, as will pressing **F10**, then **E**, or loading a program file.

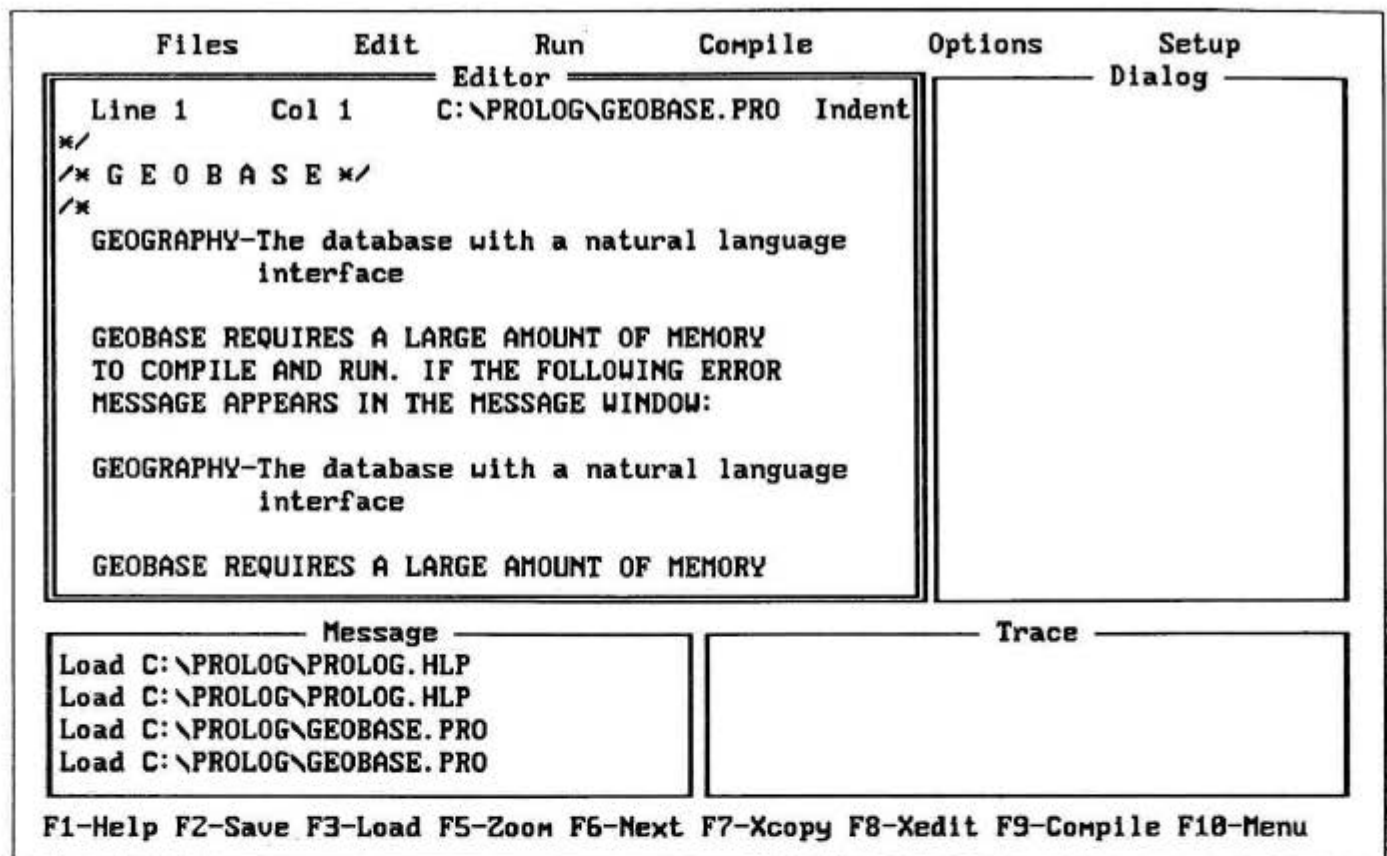


Figure 3-1. *The active Editor window*

The blinking cursor appears in the window, showing you where the editing action can start taking place. This is typically at line 1, column 1, as noted by the status line at the top of the Editor window. If you move the cursor to some other position in the file (as will be explained), leave the file to use other menus, and then return to the Editor window, the cursor will jump back to the position where you last left it.

Leaving the Editor

Pressing ESC or F10 will jump you back out of the editor to the main menu, and pressing the ALT-X hot key will jump you out of the editor and out of Prolog altogether.

Saving a File

To save whatever file you're working on in the editor, you can use hot keys, the File menu, or just press F2 to save the file to disk under the current name.

Help in the Editor

The editor has its own set of help information. While in the editor, press F1 to see the Help index, as shown in Figure 3-2. This index will open away from the position of the cursor in the editor. You can then choose any one of these subjects by pressing its first letter or by highlighting it with the cursor-arrow keys and then pressing ENTER. The information about commands in that particular area will then be listed on the screen.

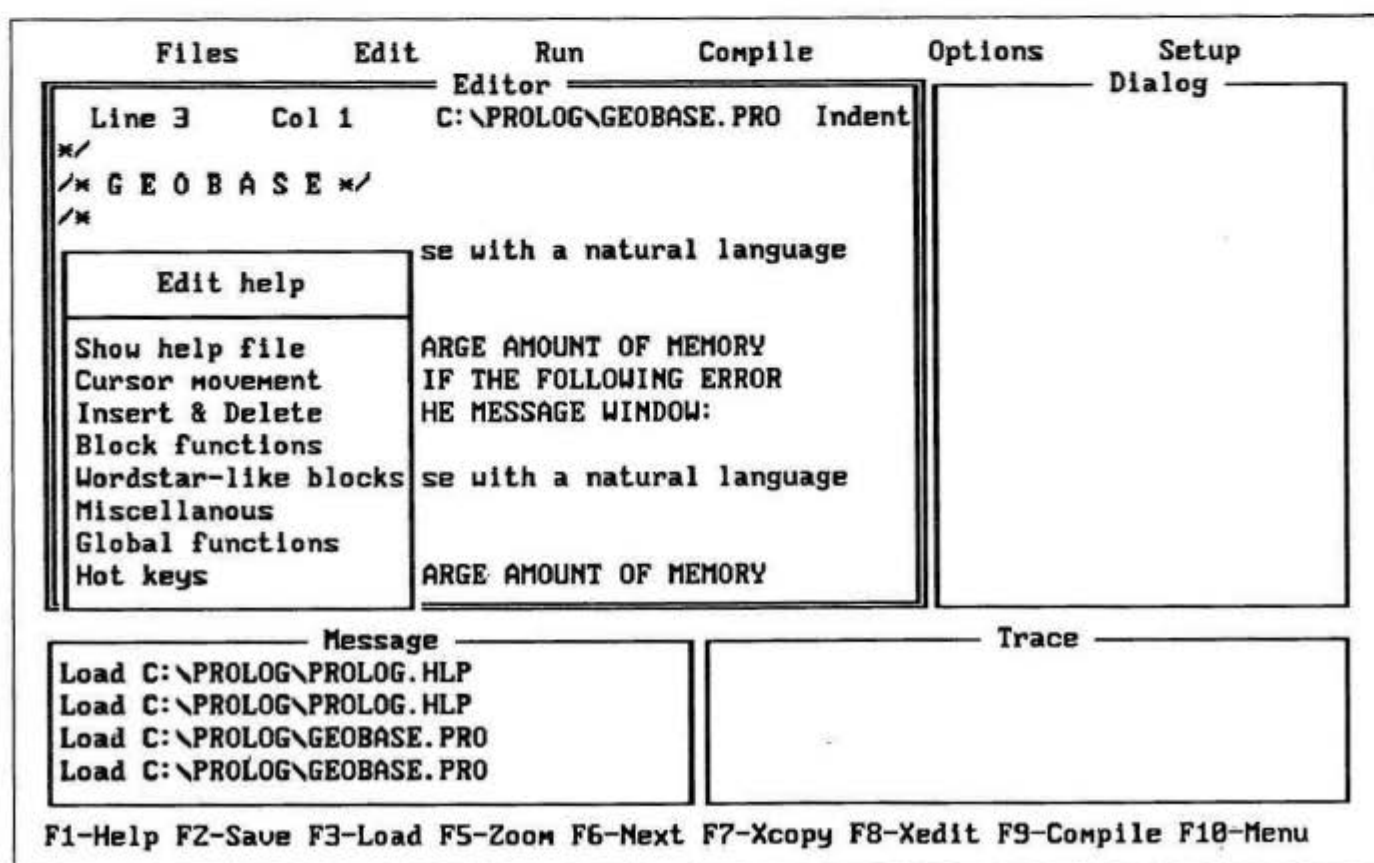


Figure 3-2. The Help index in the Editor window (press F1 to see this)

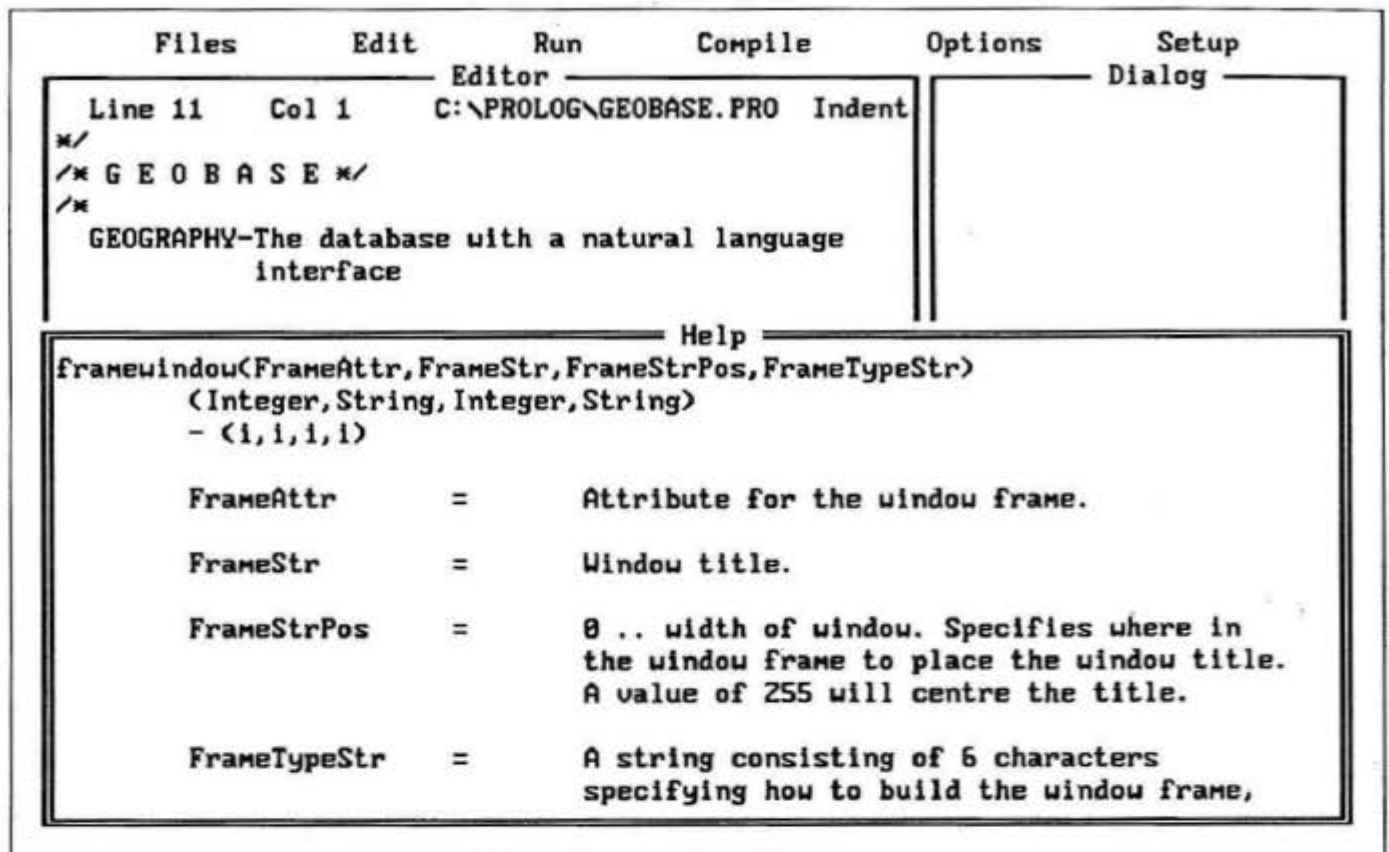


Figure 3-3. Typical page from Help file (press SHIFT-F1 in the editor)

If you press SHIFT-F1 (both the SHIFT and F1 keys at the same time), you can see an entire Help file listing the Turbo Prolog predicates, options, and just about all the details you might need while programming. Figure 3-3 shows a typical page of technical detail from this file, which you can also reach by choosing "Show help file" from the F1 Help index.

In this way you don't even have to have a manual at hand — all of the major commands are on line. Using the Search command, you can quickly locate any predicate that is in question.

Function Keys

As Figures 3-1 and 3-2 show, an active Editor window also cues a list of function-key definitions at the bottom of the screen. The following describes them in more detail.

- F1 is the help command key.
- F2 will save the file in the window to disk, using the name on the status line at the top of the window.
- F3 will load a new file into the window, asking you which file you want to load.
- F4 is not listed on the bottom line, but will start the *search and replace* function, asking you which information you want to look for, and then what to put in its place.
- F5 will “zoom” the Editor window so that it fills the entire screen, a great way to avoid enlarging it with the Setup menu just to see long code lines, then having to change the configuration and shrink it again when you need to see the other three windows. Pressing F5 again will zoom the window back to its original size.
- F6 will move the “active” window status to the next window, cycling through the Dialog, Message, and Trace windows, then back to the Editor window. This allows you to resize the windows easily.
- F7 is called Xcopy for auxiliary copy. This opens the Auxiliary Editor window, pulling in a separate file, and lets you mark the beginning and end of a block in that file (by pressing F7 again, in the correct positions), and then copy that block into the main Editor window. Before you begin this command, place the cursor where you’ll want the block to be inserted in the main Editor window. The block will be copied immediately after you mark the end of the block in the Auxiliary Editor window.
- F8 is called Xedit for auxiliary edit. Press this key to open the Auxiliary Editor window, as shown in Figure 3-4, where you can then load and edit other files.
- F9 will attempt to compile the file in the Editor window to memory.
- F10 will jump from the Editor window to the main menu, leaving the editor inactive.

Cursor Movement Commands

After you load a file into the editor or type some lines of your own for a new file, you will almost certainly need to move the cursor around to

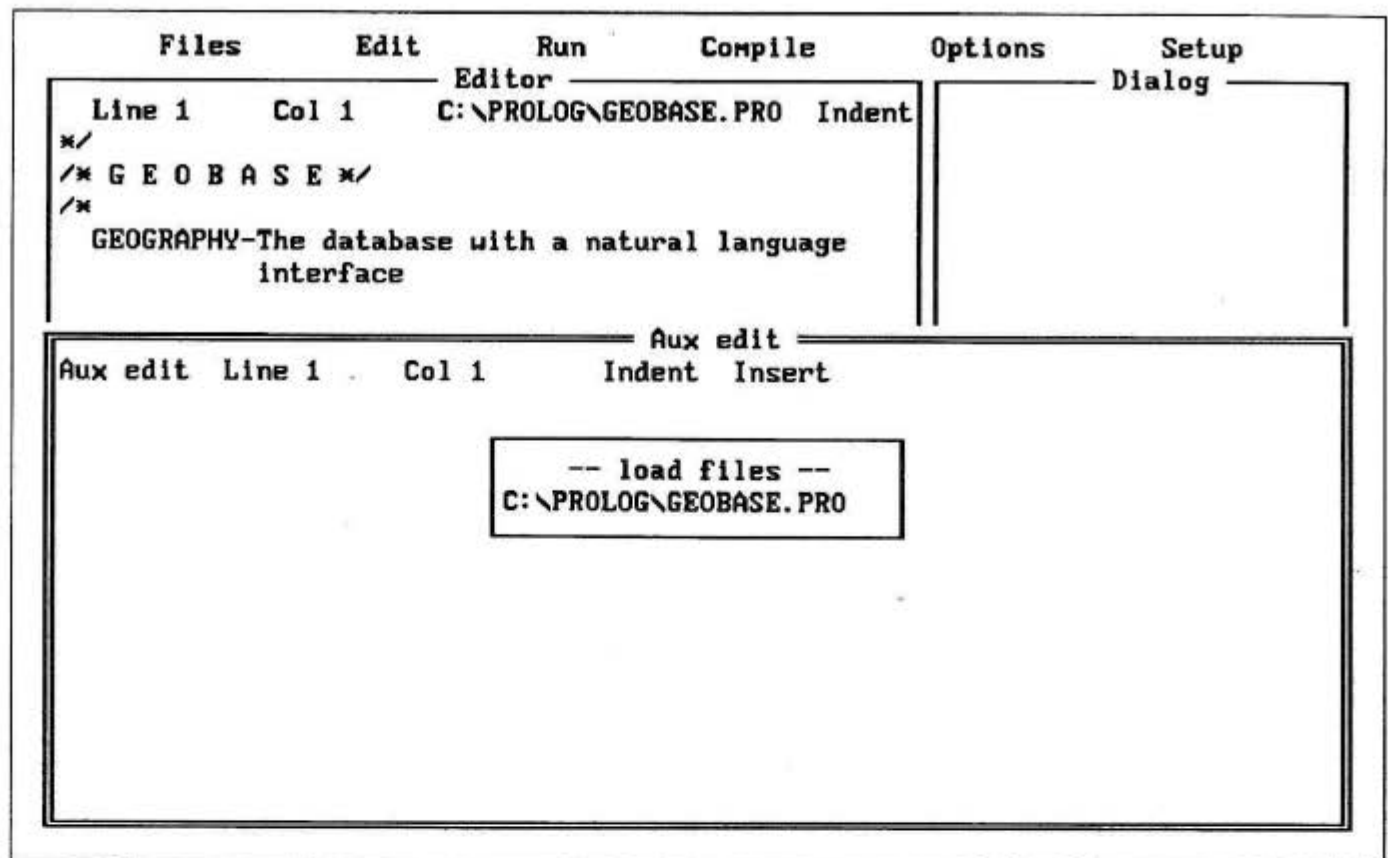


Figure 3-4. The Auxiliary Editor window (press F8 in the editor)

select lines to delete, places to add comments, and so forth. There are quite a few editor commands for moving the cursor; they divide into two sets, as shown in the Cursor-movement help display in Figure 3-5.

The leftmost column lists the operations. The center column lists one set of key actions that execute those operations; the rightmost column lists another set of key actions for many of the same operations. Whether you move the cursor a line up by pressing the up arrow or by pressing the CTRL and E keys simultaneously, the effect is the same.

The first set of commands is dominated by the cursor-arrow keys, along with the PGUP and PGDN keys. The second set (along with a few that have slipped over into the center column from this set) is dominated by WordStar-style commands that combine the CTRL key with one or two other keys to make a *key combination*. If you know WordStar or SideKick, you already know most of these key combinations, and if you learn them here you can take that knowledge to those other programs.

Experimenting with the cursor-movement commands is really the only way to get used to them. You'll soon discover, if you are typing in

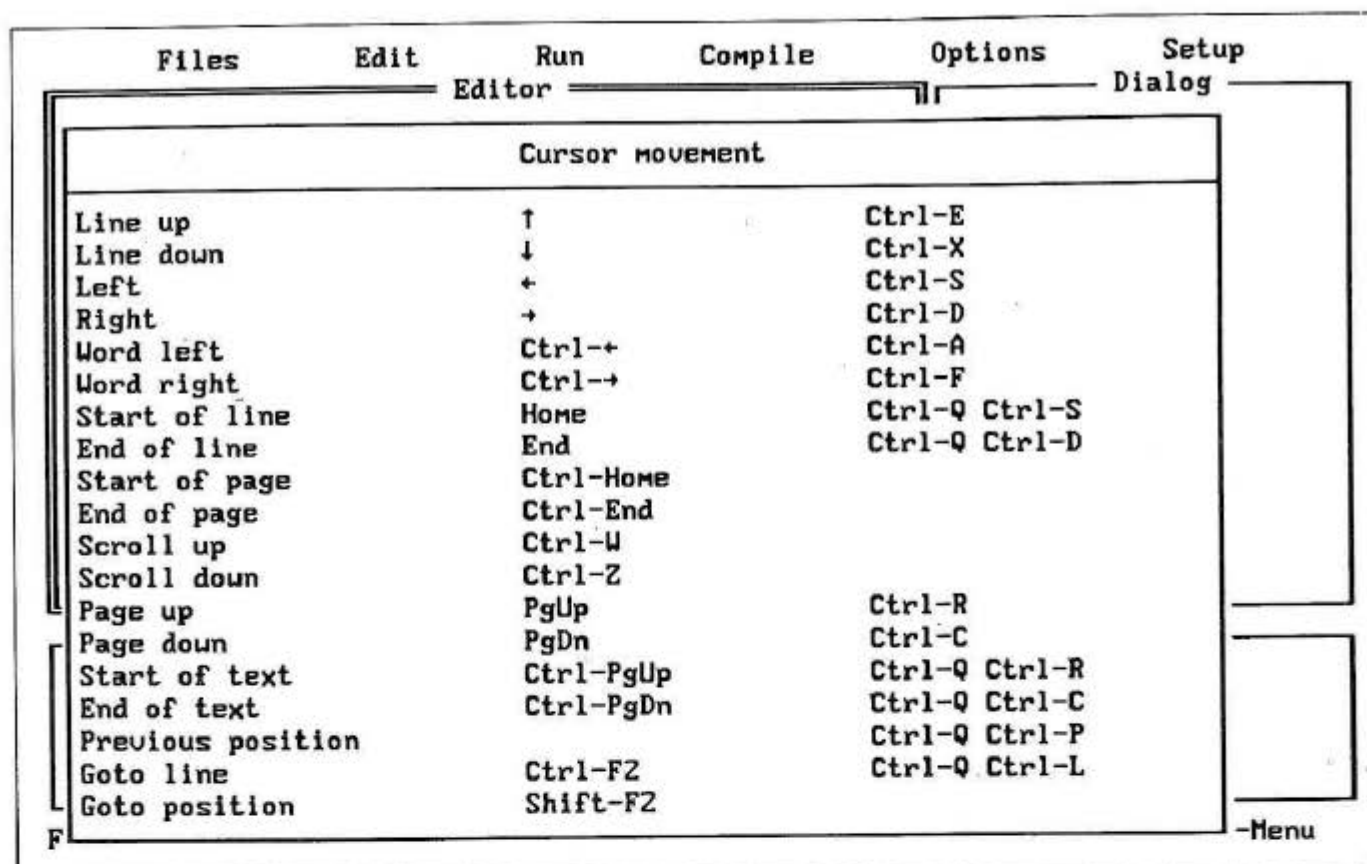


Figure 3-5. The Cursor-movement help display in the editor

text and then inspecting it with cursor movements, that there is no word wrap in the editor. That is, when you type beyond the right margin of the window, the words don't automatically leap to the next line on the far left as they would with most word processors. Instead, they keep scrolling out to the right. (But CTRL-QW drops you into text mode, allowing word wrap at the edge of the window.)

Inserting and Deleting Text

Putting in new text and deleting text you don't want are fundamental editing functions. Figure 3-6 shows the Help menu for inserting and deleting. There are also two sets of commands here, though only two pair have the same functions.

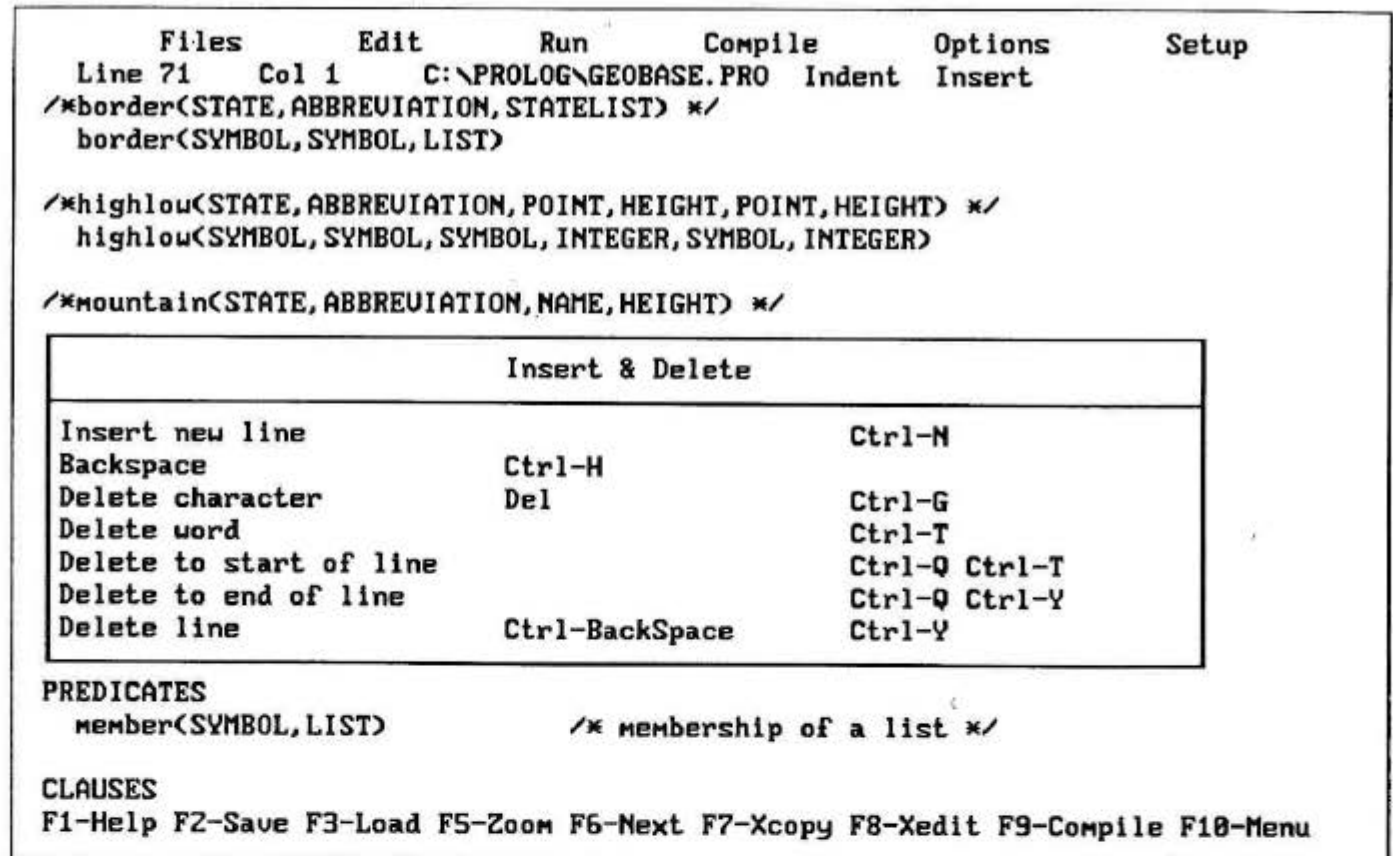


Figure 3-6. The Insert and Delete help display in the editor

Insert and Overwrite Modes

You may have noticed that on the editor's status line, on the far right, is the word "Insert" or the word "Overwrite." The editor can be in either of these modes. You can toggle back and forth between these modes by pressing CTRL-V or the INS (Insert) key.

While in *Insert mode*, anything you type will be added to whatever is already in the window. If the cursor is on top of a character, that character will be pushed ahead while the new character is "inserted" where the old character was.

In *Overwrite mode*, new characters fall on top of and substitute for the old characters, wiping them off the window and out of the file being edited.

Indent Mode

When you press ENTER to move to a new line for entering text, the cursor doesn't always return to the first column of the next line. Many times you'll see it move to the same column of the first character on the line above. This is called *auto-indenting* and is very helpful in programming, where many lines need to be indented the same amount within a section to keep the program neat and readable. The status line of the Editor window has the "Indent" sign on when auto-indenting is active. You can toggle the auto-indent feature on or off by pressing CTRL-Q-I.

Block Functions

In many circumstances it is handier to move, copy, or delete an entire block of text rather than work by a single word or line at a time. To do this, you need *block functions*, and the built-in editor has a bundle of them. Figure 3-7 shows the regular block functions of the Prolog editor. For example, to move a block, you would move the cursor to the beginning of the block, press ALT-F6 (and highlighting would begin), move the cursor to the end of the block (which would be entirely highlighted), press ALT-F6 again, move the cursor to the place where you want to move the block, and press ALT-F6 a final time. The block will pop to the new position. At each step, the bottom lefthand corner of the window would hold a message telling you what to do next. Figure 3-8 shows one of these messages.

The list in Figure 3-7 points out that you can copy a block to another position, to the printer, or even to a disk file, move a block, change the case of a block (the capitalization of its letters), delete a block, and even undelete a block.

Search and Replace

The Block functions menu also contains the Search and Replace functions. These let you hunt for specified phrases, numbers, or

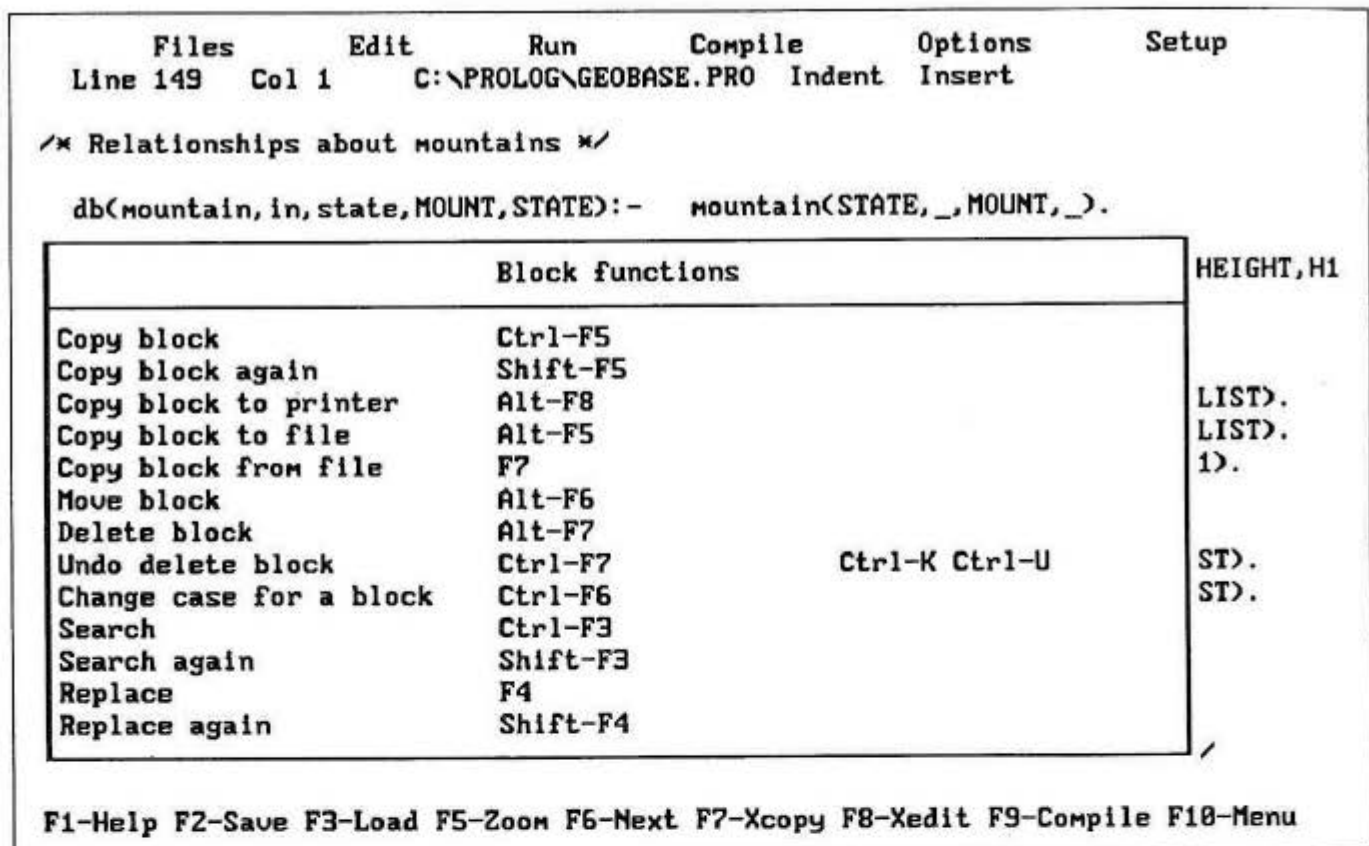


Figure 3-7. The Block functions help display in the editor

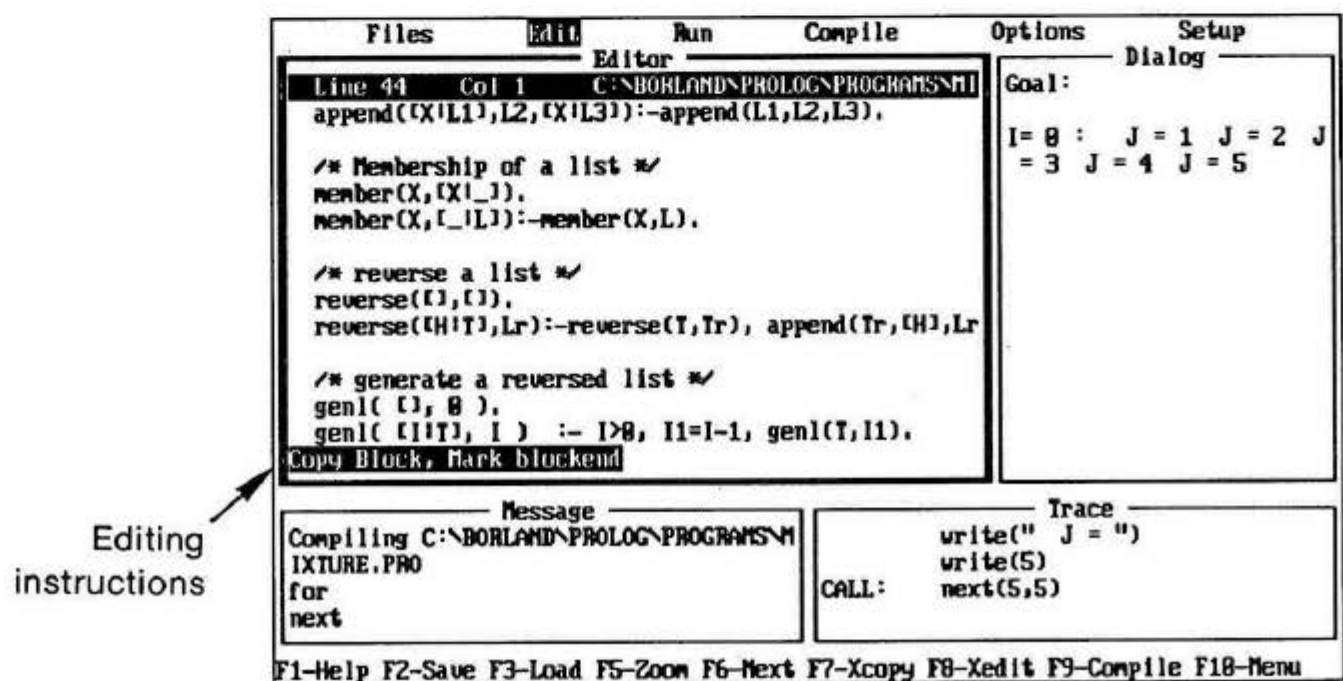


Figure 3-8. Moving a block (with instructions at the bottom)

characters in a file, and are useful in many programming tasks. For instance, if you changed the name of a variable (explained later in the book) that was used in several places in a program, you might want to *search* for that variable (to find where it was used) or *replace* that variable (to automatically and thoroughly change the old name to the new name in every occurrence).

Just as with the block functions, messages for these operations occur at the bottom lefthand corner of the window, asking you what to search for and what to put in its place. After you answer these questions, you'll be asked

Global/Local (g/l)?

to see if you want to work on all occurrences (global) or just the first one (local). Press **g** for global and **l** for local. Then you'll be asked if you want to

Prompt before replace (y/n)?

to see if the computer should ask you for confirmation before replacing any particular occurrence. Finally, if you do ask for prompting, you'll see

Replace (y/n)?

along with the cursor resting on an occurrence of the specified phrase.

WordStar-like Block Functions

Figure 3-9 shows another set of block commands. These follow the WordStar block functions, with plenty of CTRL combinations with other keys. The first step in any of these functions is to mark a block by pressing CTRL-K-B when the cursor is at its beginning and CTRL-K-K when the cursor is at the block's end. (You can press the first CTRL and K together and then release the CTRL before pressing the second B or K, if that is more comfortable.)

Files	Edit	Run	Compile	Options	Setup
Line 152	Col 1	C:\PROLOG\GEOBASE.PRO		Indent	Insert
/* Relationships about mountains */					
Wordstar-like blocks					HEIGHT, H1
Set block start			Ctrl-K	Ctrl-B	
Set block end			Ctrl-K	Ctrl-K	
Hide/Show block			Ctrl-K	Ctrl-H	
Goto block start			Ctrl-Q	Ctrl-B	LIST>.
Goto block end			Ctrl-Q	Ctrl-K	LIST>.
Copy block			Ctrl-K	Ctrl-C	1>.
Move block			Ctrl-K	Ctrl-U	
Delete block			Ctrl-K	Ctrl-Y	
Change case for a block			Ctrl-K	Ctrl-E	ST>.
Copy block to printer			Ctrl-K	Ctrl-P	ST>.
Copy block to file			Ctrl-K	Ctrl-W	
Copy block from file			Ctrl-K	Ctrl-R	
Search			Ctrl-Q	Ctrl-F	
Search again			Ctrl-O		
Replace			Ctrl-Q	Ctrl-A	/
Replace again			Ctrl-L		
					0-Menu

Figure 3-9. The WordStar-like Block functions help display

Once a block is marked, you can move, copy, or delete it. You can even unmark it by pressing CTRL-K-H.

The search and replace functions have other keys in this set too, but they will prompt the same operations.

Miscellaneous Commands

The list of commands in Figure 3-10 combines some commands that have already been mentioned (toggling Insert and Indent modes, getting help, starting the auxiliary editor) with a few more hard-to-classify commands. Both F10 and CTRL-K-D will jump from the editor to the main menu, while three different commands can change the case of a word.

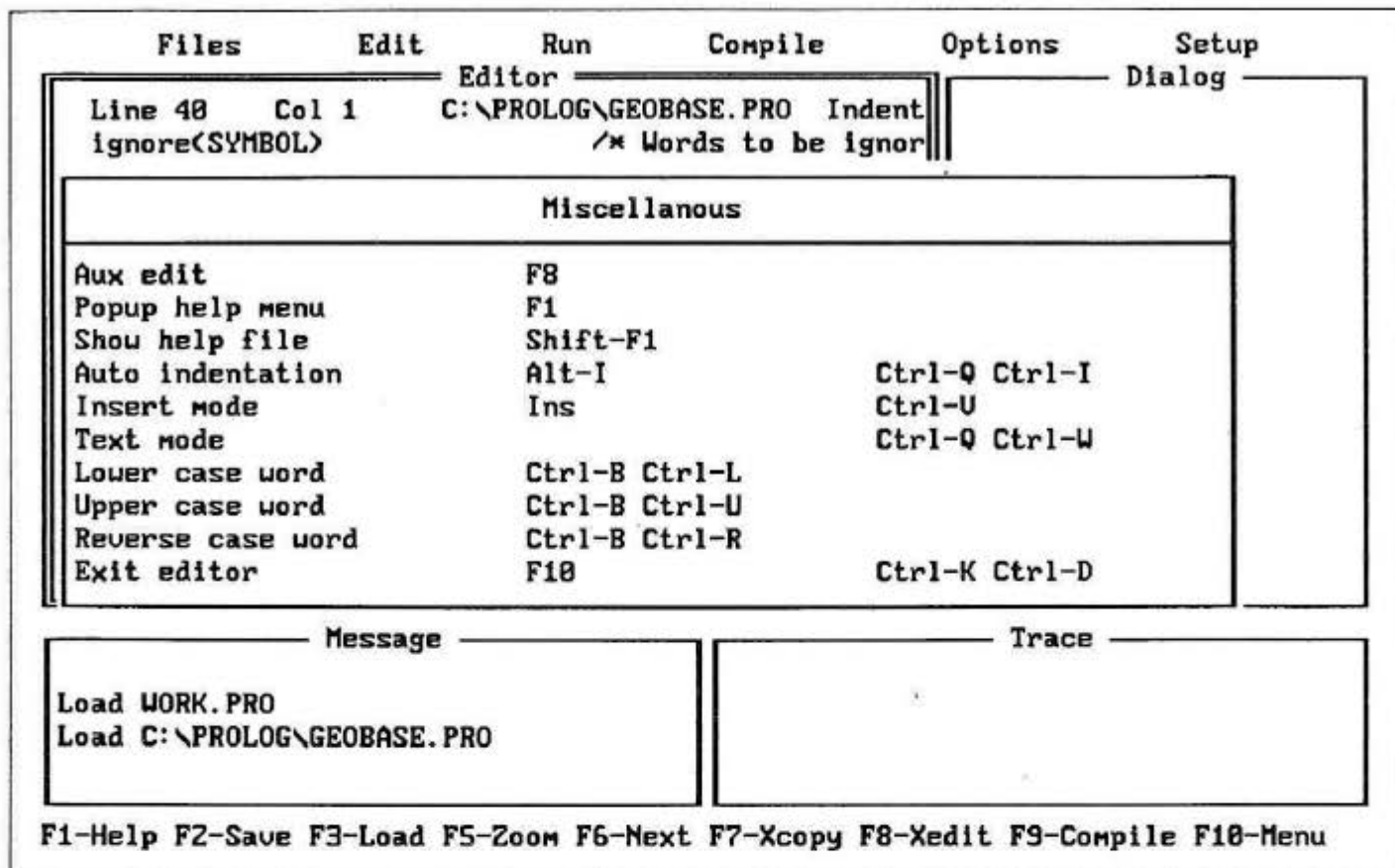


Figure 3-10. The Miscellaneous functions help display in the editor

Hot Keys

The hot keys are global in Turbo Prolog. Choose this option in the editor's Help index and you'll see the same list as shown in Figure 2-19. These keys include combinations for quickly saving a file from the Editor window or for leaving the editor and pulling down another menu all in one swoop.

Part Two

Every Turbo Prolog program is built around a few basic concepts. Part Two explores these concepts, beginning with facts and rules. The section then moves on to topics such as backtracking, arithmetic, and structures (functors, lists, and strings). By the time you complete this section, you'll be writing powerful Turbo Prolog programs of your own.

4 Facts: Objects and Relationships

If you feel comfortable with Turbo Prolog's *environment*—the windows, menus, and editor—you're ready for the language itself. This chapter will tackle the first step in using Prolog: understanding logical clauses about objects and relationships. Because this is the introduction to a whole new style of programming and logic, this chapter will move carefully and slowly. If you're in a hurry, don't get tense; the pace will pick up after the introduction to the logical elements.

Procedural Versus Declarative

Most of the languages that have been used on microcomputers—BASIC, Pascal, Modula-2, C—have been *procedural*. Such languages let the programmer tell the computer what to do, step by step, procedure by procedure, to reach a conclusion or perform a function.

Prolog is not procedural. It is *declarative*. If you haven't used a declarative language before, learning Prolog is going to take more

adjustment than just learning another procedural language would require. But the effort will definitely be worth it, because a declarative language does a lot of programming work for you. While a procedural language demands that you enter both the recipe and the ingredients, a declarative language asks only for the ingredients and the goal product. You declare the situation to work with and where you want to go. The language itself—the Prolog compiler, in this case—does most of the work of deciding how to reach the goal.

For instance, if you are a BASIC programmer who wants to learn Pascal, you soon notice many constructs that are similar from language to language. You recognize assignment statements, IF-THEN choices, FOR-NEXT loops, and the like. But if you're moving from BASIC to Prolog, you have to take a bigger step. You won't recognize many of the text structures. A Prolog program may look just like a database of facts—lots of parentheses on unrelated lines without rhyme or reason. Don't worry; you'll find it as easy to read as BASIC after a little practice. And as you progress, you'll discover some practical commands that are quite similar to advanced statements in Pascal or other procedural languages. You'll probably also be delighted at how much you can accomplish with just a few lines of code, laying down facts and dictating the form for input and output.

Objects and Relationships

Prolog at its simplest deals with objects and the relationships between them. You may even hear it called an object-oriented language. An *object* isn't necessarily something tangible. It can be anything you can represent symbolically on a computer.

Prolog has been more popular in Europe than in the United States for some years (though the advent of Turbo Prolog has certainly created a bevy of Prolog programmers in the United States in the past couple of years). It was invented in France and has been widely used from Hungary to England. That European affiliation might be the reason that members of a royal family are frequent demonstration guests in older Prolog texts.

Here are some examples of objects:

charles
philip
diana
elizabeth
henry

Here are some relationships that are useful when considering those objects:

king
queen
prince
princess
mother
father
son

If you're not a royalist, here's another set of objects and relationships. The objects are

cow
bat
iguana
redwood
fern
ibm pc
apple macintosh

and the relationships are

animal
mineral
vegetable
mammal
reptile
computer

You may have noticed that none of these words are capitalized. That's not just a slip; they are supposed to start with a lowercase letter. Beginning a word with an uppercase letter has a particular meaning in Prolog that will be explained later when variables are discussed. For now, stick to lowercase typing. The examples of

objects and relationships so far have been pretty clear-cut, real-world examples. You could also have objects such as:

freedom
theft
bargor

What's a "bargor"? Webster never heard of it, and Prolog doesn't care. Prolog deals with objects as logical entities, and never really does "know" what they are. You understand the reference of many object words because of a lifetime of learning and associations. The computer doesn't have this and doesn't need it.

Relationships can be defined in the same way—they can be familiar or tangible, but they don't have to be. For instance, these could be relationships:

greater_than
swomblier
teth

Choosing object and relationship words that have some definite meaning and reference may be more practical, but it's not necessary. It is good to remember that Prolog is treating these words as logical units, dealing with the positions and relationships that you set up in the Prolog file and syntax, without understanding what they are in the outside world.

Syntax

The *syntax* of a programming language is the body of rules that governs a source file: what words are acceptable to use, what position those words must occupy, where the punctuation goes, and so on. Even if you understand the theory of a language, you won't get anywhere if you don't stick carefully to its syntax. Computers are built to understand exactly what you said, not what you might have meant. With enough power they can try to guess your meaning in

some cases, but even then they are doing that by carefully following rules that someone wrote down as a program, with all the punctuation in the right places and the appropriate words capitalized or left lowercase.

Several types of syntax exist in Prolog. Different implementations of the language sometimes follow noticeably different rules. Still, many of the constructs in the language work the same way, even if they come in different clothing.

Turbo Prolog contains most of the features and syntax of the Prolog described in W. F. Clocksin and C. S. Mellish's book, *Programming in Prolog*, mentioned in Chapter 1. This is good reading if you're interested in learning more about Prolog after you finish this book and the tutorial in the Borland Turbo Prolog *Owner's Handbook*. The Clocksin and Mellish text used a particular style of written Prolog, with a particular set of punctuations and key words. That style has since become the "unofficial" standard with many Prolog users.

Facts

The first way to combine an object and a relationship is to use them to define a fact. Turbo Prolog's syntax wants you to write facts in the following way:

relationship (object)

Note that the object is inside the parentheses and that the relationship precedes it. This object has that relationship. When written in this form, the relationship is known as the predicate and the object is known as the argument. You could grab some objects and relationships from the previous two lists and make up the following facts:

mammal (bat)
prince (charles)

The second fact means that the object charles has the prince relationship. The English translation is necessarily vague, because the logic of the Prolog fact does not specify that "charles was a prince," or "charles saw a prince," or even "charles is definitely not a prince."

As you use Prolog, you need to keep in mind what the relationships you use in your program stand for—Prolog can't do that for you. Your programs will make sense only if you are consistent throughout a single program about the meaning of a given relationship. It is entirely possible to write a long program, and assume that a relationship means one thing in one part and something else in another part. Prolog will assume they mean the same thing in all parts, which could throw your logic and results out of whack.

It helps sometimes to use relationship words that more closely approximate what you mean. For instance, if you wanted to indicate the fact that charles is a prince, you might use this relationship with him as an object:

is_a_prince (charles)

The underlines indicate to the computer and compiler that this is all one long "word" for a relationship.

Truth and Garbage

The old computer acronym GIGO (garbage in, garbage out) definitely applies to Prolog. The compiler has no way of ascertaining if a fact is true or false in the real world. If you put the fact

king (diana)

into the program, Prolog will accept it and use it for future logical reasoning. Whatever you tell Turbo Prolog is a fact it will accept as a fact. You have to be responsible for verifying your facts to what-

ever level you need. Sometimes you might want to use blatantly false facts. If you are trying to decide what might happen in a certain situation, for example, you may want to use “what-if” facts that aren’t necessarily proven, demonstrated, or even close to the present truth.

Here are some more facts from the second list of objects and relationships mentioned previously:

```
animal (cow)
animal (bat)
animal (iguana)
vegetable (redwood)
vegetable (fern)
computer (ibm_pc)
computer (apple_macintosh)
```

Again, the underline is used to make a single object out of several words.

These facts can make the foundation of a Prolog program. Before you enter them in the Editor window, though, there is one more thing you should know: how to add comments to a program and what the section divisions of a Turbo Prolog program are.

Comments

The Turbo Prolog compiler turns your source file into an *object file*—a program that your computer can run directly. It converts each bit of text into machine-language code. But sometimes you’ll want the compiler to ignore certain parts of the text. For instance, you might want to start the text of a program source file with the words:

First Example Turbo Prolog Program

If you enter only those words, the compiler will be confused while

trying to interpret them. So turn them into a *comment* by surrounding them with comment marks like this:

```
/* First Example Turbo Prolog Program */
```

When a section of text is preceded and succeeded by slash-asterisk marks, the compiler ignores it. This lets you add text to a program to explain what it is, what it does, and how it does it. You might use comments, for instance, to explain the meaning of the relationships you use in your facts.

In version 2.0, there is a second way to add comments, a method that lets you put a comment on the same line as the program code it refers to. Any characters after the percent sign on the end of the line are treated as a comment, as long as the percent sign is not inside a character string, such as "95% pure."

Program Divisions

Most Turbo Prolog programs are organized into four major sections:

- Clauses
- Predicates
- Domains
- Goal

Although not all of these sections need to be present in all programs, you should be familiar with all of them, and will be using all of them in this book. Other sections are sometimes added to include special constructs in a program. These will be described later in the book.

Clauses

The facts that you build out of your objects and relationships are listed in the Clauses section. The Clauses section also holds your program's "rules," which will be introduced in Chapter 5.

Predicates

Predicates are the relationships. (The term predicate is borrowed from formal logic, one of the initial seeds for Prolog.) Whenever your program is going to use a particular predicate in the Clauses, you need to formally "declare" it in the Predicates section. This scheme lets the compiler keep track of the predicates and what sort of information they are supposed to work with, and aid with writing an organized program. These topics will be explained as we work through some example programs.

Domains

Turbo Prolog wants one more level of explanation before you call a program complete. You need to tell it about the arguments your predicates will use. To keep computer memory use to a minimum and to make programs easier to debug, Turbo Prolog has a lot of *type checking*. (If you're a Pascal programmer, you've seen type checking before.) It wants to know in advance what type of thing an argument can be, such as a string of characters, a simple number, or a scientific number.

Goal

This is the section that lets Turbo Prolog know what you want to find out or what you want the computer to do with the information

you've provided in your program. Normally a program will have at least the Predicates and Clauses sections, but it is possible to have a program that is only a Goal section. The Goal section is one place where Turbo Prolog differs from Clocksin and Mellish's Prolog. Clocksin and Mellish keep the goal outside of the program—an external goal. Because Prolog can be used *interactively*—you can “query” a program while it is running—it is common to run a program and then wait for it to ask you for a goal. You type the goal into a terminal and then wait to see what Prolog reasons and gives you back.

Turbo Prolog can also work with an external goal, but it provides a Goal section to let you run programs noninteractively. You can build the goal right into the original code so that, as soon as you run the program, it will start working toward the desired solution.

The First Program Example

Run Turbo Prolog, get into the editor, and type this first line:

```
/* Example 1 -- Animal, Vegetable, or Computer */
```

Then press ENTER. The screen may scroll while you type the first line, but it will return to the initial position when you press ENTER.

Next type the words:

```
domains  
predicates  
goal  
clauses
```

This is the skeleton for a program.

After the word “clauses”, press the TAB key to indent the actual clauses, and enter the facts as shown in Figure 4-1, ending each fact with a period and pressing ENTER. The period tells the compiler

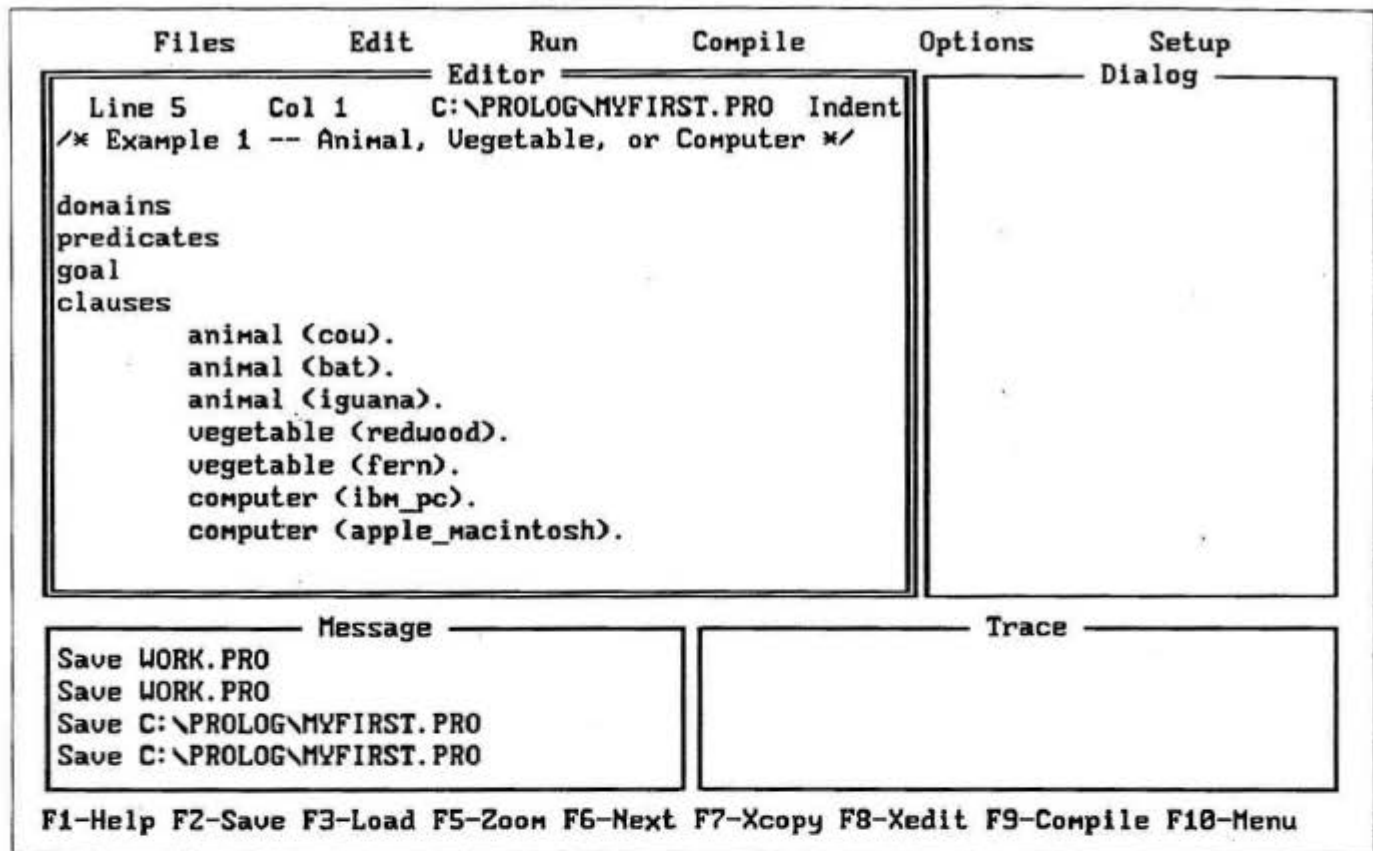


Figure 4-1. First Program Example in progress: clauses

that this clause is done and that it should prepare for another clause or the end of the file. Pressing ENTER moves the cursor to the next line. If auto-indent is on, you'll see the cursor stop at the same horizontal position as the beginning of the previous fact. If auto-indent isn't on, press CTRL-Q-I to toggle it on.

Now that you have some clauses, you need to tell Turbo Prolog about the predicates (relationships) you're using in those clauses. Open a blank line just after the word "predicates" and tab in. Add these lines:

```

animal (thing)
vegetable (thing)
computer (thing)

```

on three lines, as shown here. (The predicate declarations don't need to end with a period like the clauses do.) These declarations have informed Turbo Prolog that the predicates animal, vegetable,

and computer all work with a single argument, and that they all have the same “type” of argument: a “thing”. Not surprisingly, the computer now wants to know what kind of things they are. That’s what the Domains section is for.

Tab in on the line after the “domains” title and add this declaration:

```
thing = symbol
```

with a space after “thing” and another space after the equals sign. Now you’re done typing.

A “symbol” is one of the official “types” in Prolog. I’ll describe the others later. With clauses entered and both predicates and arguments declared, the program is coming close to completion.

Save What You’ve Got

As with any computer use, it’s a good idea to frequently save to disk whatever it is you’re working on. If you just press F2, this program will be saved under the name on the title line—WORK.PRO in my example. If you want to save it under another name, such as MYFIRST.PRO, press the ALT-F hot-key combination (to jump from the editor to the Files menu), and press W for “Write to.” Type the file name, press ENTER, and you’re done. (You don’t have to use capital letters and you don’t have to use the PRO extension—Prolog automatically adds it. Press E to return to the editor, where you can resume your programming task. When you are inside the edit window and use the F2 option for saving, you are automatically returned to the editor window. Either way, you’ll see a Message window note confirming the save action.

Compiling the First Program Example

Now you’re ready to compile the program, which is shown in its entirety in Figure 4-2. You could press ESC to return to the main menu, then use the Compile command from that menu. To hurry

Files	Edit	Run	Compile	Options	Setup
Line 4	Col 23	C:\PROLOG\MYFIRST.PRO	Indent	Insert	
/* Example 1 -- Animal, Vegetable, or Computer */					
domains					
thing = symbol					
predicates					
animal (thing)					
vegetable (thing)					
computer (thing)					
goal					
clauses					
animal (cow).					
animal (bat).					
animal (iguana).					
vegetable (redwood).					
vegetable (fern).					
computer (ibm_pc).					
computer (apple_macintosh).					
F1-Help F2-Save F3-Load F5-Zoom F6-Next F7-Xcopy F8-Xedit F9-Compile F10-Menu					

Figure 4-2. Completed First Program Example: before first compile

things up even more, you may just press the F9 key, as is shown on the bottom line of the Prolog display.

Unfortunately, the compile won't succeed. The Message window will tell you that it began, but an error message in the Editor window will inform you that a

419 Syntax error in clause body, predicate call expected

mistake got in the way. This is shown in Figure 4-3. The error message even points out where the error happened, placing the cursor on the error spot.

Syntax Checking

Because the Goal section did not have anything in it, the compiler thought it was another predicate declaration in the Predicates section. This built-in syntax checking is an enormous help during

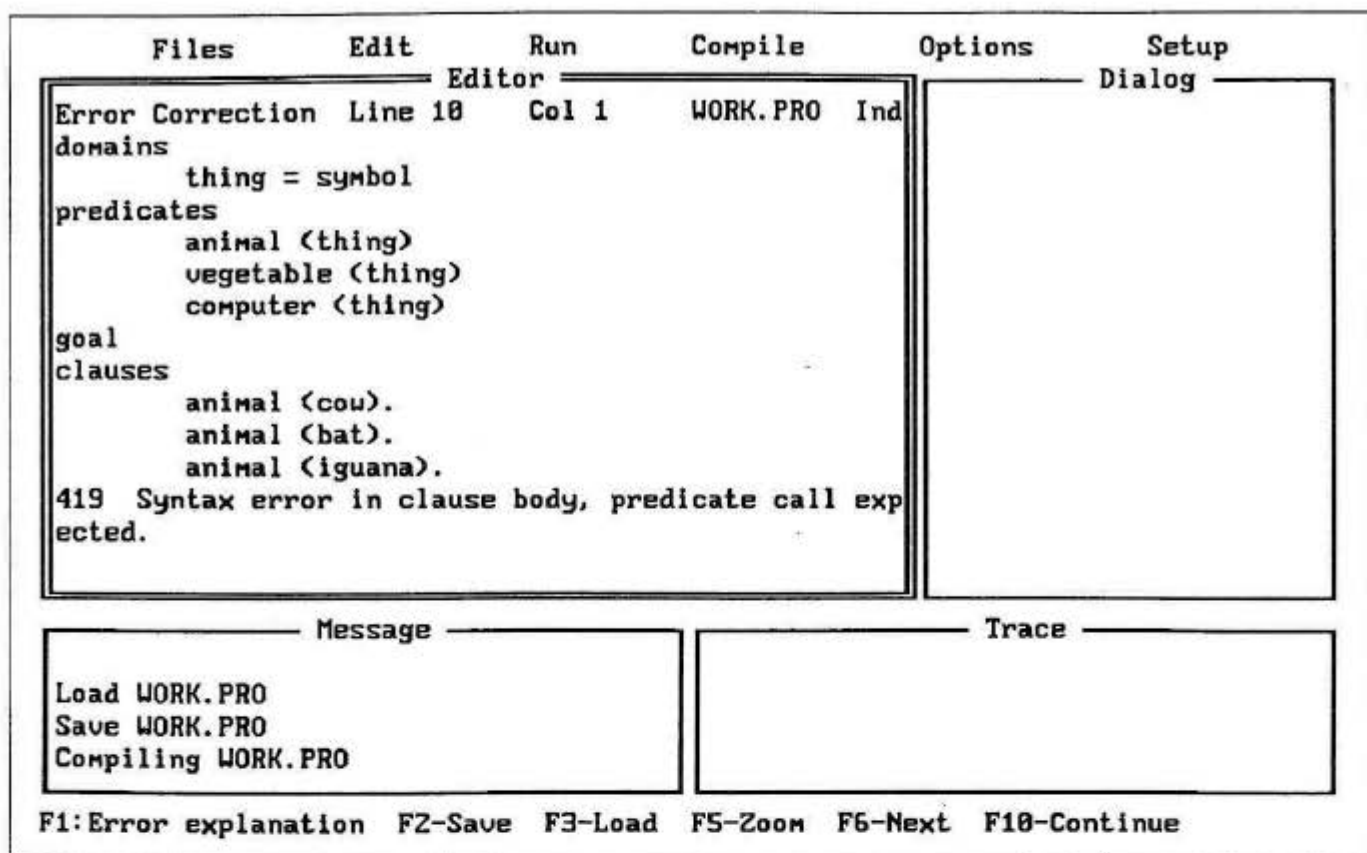


Figure 4-3. Syntax error from first compile attempt on First Program Example

program development. If you want more information on the error, Prolog can sometimes give you more detailed explanations at the touch of the F1 key. (In this case, there is nothing else available.)

Debugging the First Example

Delete the “goal” line and try compiling again. The syntax checker will have put you right back in the editor after that last mistake, so you can just eliminate the line, save your file, and press F9.

Running the Example

After the compilation is successful (which it should be now), press R to invoke the Run command. Your program, which was compiled into memory and is waiting in RAM, will run within the Turbo Prolog environment. If you have other errors, try to eliminate them in the same way the goal syntax problem was just found and remedied. If you still have trouble, back up and type the program in from the start.

If you want to take a shortcut to a finished running program, try the ALT-R hot key instead of F9. This will compile and run the program in one shot.

When a program with an external goal (one without a Goal section) runs, it uses the Dialog window to let you provide a goal to the program. You'll see the word

Goal:

appear in that window. At the Goal: prompt, you can type a query for the program and receive an answer. In fact, you can then continue to query the program, providing goal after goal for it to pursue until you wish to stop running the program.

Goals and Questions

Although the goals you set for your Prolog programs may eventually be as complex as deciding on medical treatment or showing a graphics simulation of an aircraft in turbulent flight, the goals for this first example have to be considerably less exalted. There just isn't enough information in the Clauses section for Prolog to squeeze much out of the program. If you ask questions of a Prolog

program that don't pertain to the knowledge that you've provided it (the clauses of facts, for example), it will not merely be dumb-founded, it will attempt to reason about your query using the facts it has, and therefore will produce inaccurate results. If a Prolog program knows about minerals and you ask it about sports, it will almost always return the answer "No" to any question you ask, even if the answers might be true in the real world. After a few examples, you'll see why this is so.

Later programs in this book will have internal goals. This first program deals only with *external goals*—goals that you enter in the Dialog window at the Goal: prompt. That window is active and will let you type in information by using many of the same editing commands that the Editor window understands.

Checking Facts

The least complex goal just checks a fact. If you enter a fact in the Dialog window, Turbo Prolog will take it as a question and will tell you if that fact can be found in its list—the clauses section. For this example program, type **animal (bat)** and press ENTER. You have asked or queried, "Does the bat object have the animal relationship?" in logical terms, or "Is a bat an animal?" in practical terms. You'll immediately see the answer—the word "Yes" appears, as shown in Figure 4-4. The program will still be running and will then ask for another goal.

Try the goal **computer (ibm—pc)** and you'll see the answer: "Yes." However, the goal **vegetable (ibm—pc)** will come back as "No."

In these cases, the Prolog compiler searched through its known list of facts to see if it could prove, using those facts, the truth of the query. Any fact that it can find or logically build from its list will be thought of as true and Prolog will return "Yes." Any fact that is not there or that can't be logically induced from the clauses is false, whether or not that query is in reality true in the outside world, and Turbo Prolog returns "No." Neither "Yes" nor "No" means that anything is verifiably true or false in the outside world. If the program's information is wrong, the answers will often be wrong.

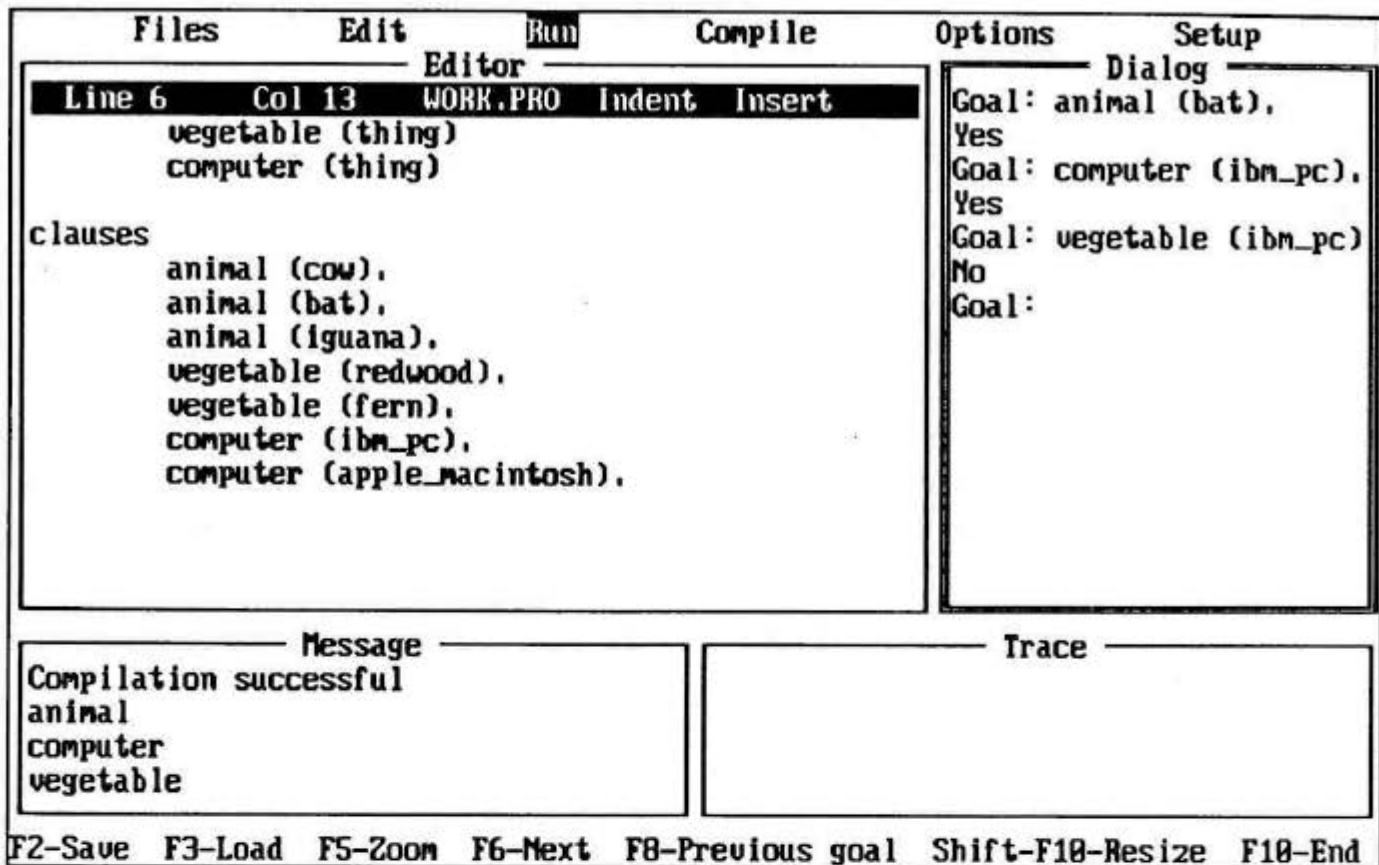


Figure 4-4. Queries to a program with no internal goal: Dialog window

Stopping a Program

If you want to stop the program and leave the Dialog window, press ESC. That will get you back to the main menu. You can also use the hot key ALT-E to stop the program and jump directly back to the Editor window to add to the program or modify it. Either way, press ALT-R to resume running the program (which is still compiled in memory.) The key combination CTRL-ESC, when pressed once or twice, clears all the windows (except the Edit window) and clears your compiled program from memory.

Searching for Information

Prolog searches through the clauses from top to bottom and from left to right. In the cases of `animal (bat)` and `computer (ibm_pc)`, it found facts that exactly matched the goal. In the case of the goal

vegetable (ibm—pc), it searched through the database and couldn't find an exact match. Therefore, as far as this program is concerned, there is no way to know if an ibm—pc is a vegetable. It could be, logically. Just because an object has one relationship—as in computer (ibm—pc)—doesn't restrict it only to that relationship.

The queries really ask, "Does your knowledge tell you that the object has the relationship?" and Prolog answers from its fund of knowledge.

Variables and the Goal-Search Process

There is another way to ask questions. If you know the relationship but not the objects, you can use variables. Any object word that begins with a capital letter is considered a variable. After the capital, you can have any number of letters (upper- or lowercase), along with digits and underscores. The quickest way to see what a variable can do is to ask a question of the example program. Try

animal (X)

as a goal. The "X" is the variable. The answer will be

X=cow

X=bat

X=iguana

3 Solutions

Turbo Prolog takes the goal with the variable and compares it to the clauses. First it checks to see if the predicate is the same as in the first clause. If it is, then it tests to see if the goal and the predicate have the same number of arguments. If either of these things isn't true, Turbo Prolog gives up on the first clause and moves on to test the goal against the second clause. If both are true—the same predicate and the same number of arguments—Prolog then tests to see if the arguments are the same.

In this case, the goal argument is a variable, and it can stand for anything. So the compiler *instantiates* it to the value in the clause;

in other words, the compiler assumes for a time that the variable has the value of the argument in the clause. In the language of Prolog, this is called *binding* a value to a variable. Now the goal and the clause are in exact agreement, and Turbo Prolog can report to you in the Dialog window that it has found an instance of the fact you asked about. It tells you the instance, along with the value it temporarily set the variable equal to, and then goes back to the clauses.

Because you used a variable, Prolog will look for all clauses that might fit the goal or make the goal true. It will release "X" from the first value binding, and will move on to the next lower (or farther to the right) clause. It will inspect this clause just as it did the first clause. Eventually, Prolog will serve up all cases that allow the goal to be true. The current example yields three instances of the goal being true.

When you don't use a variable, Prolog checks in the same way to see if your query fact is true, but it does not tell you the value found for that truth. It also stops after the first truth, not checking to see how many times the fact might come up in the clauses.

5 *The Basics of Programming in Prolog*

Turbo Prolog has much more to offer than the elementary facts and variables described in Chapter 4. This chapter introduces facts with multiple arguments, compound goals, anonymous variables, rules, and backtracking, and shows how tracing can be used with them. By the time you finish this chapter, you'll see that these concepts are easier to understand than you may have thought.

If you're an old Prolog hand and know about rules, backtracking, and compound goals, you can skim most of this chapter, since it covers material that you will already be familiar with. You should, however, take a longer look at the "Tracing" section, because you'll want to know how to use Turbo Prolog's built-in Trace window and Tracing options, features not found in other Prolog implementations.

Prolog's Building Blocks

In the previous chapter, you learned how Prolog programs are made up of facts. As you recall, a *fact* is a relationship between an object or a group of objects. Relationships in Prolog are known as *predicates*, while objects are known as the *arguments* to a predicate. Arguments in Prolog are also made up of items, called *terms*, which are the basic building blocks of a Prolog program.

Terms

A term can be either a constant, a variable, or a data structure. In Chapter 4 you worked mainly with constants. A *constant* represents a specific object or value, such as

```
cow  
1392.10  
ibm__pc  
apple__macintosh
```

A constant must start with a lowercase letter (or a digit if it is numeric in nature), but from there it may contain just about any group of letters, digits, or underscores. An argument that is a constant has a set value and may not change its value during the operation of a program.

A variable, on the other hand, must always begin with an uppercase letter or an underscore. In Chapter 4, you were introduced to variables with the ever popular "X". A variable stands for an object whose value is not yet known. Variables are used in Prolog programs when you do not know the value of an argument but want to find it out. During the course of processing, a variable may become *bound* to any value or Prolog term. Once a variable is bound, the value of that variable is identical to the value (or term) to which it is bound. It is as if the variable is actually the value or term that it is bound to.

The last type of term in Prolog is known as a *data object*. Since only basics are being discussed here, the topic of data objects, both compound and complex, will be covered in Chapter 6.

Arity: Facts with Multiple Arguments

The *arity* of a predicate represents the number of arguments that a predicate has. For example, the following predicate has an arity of three:

```
mountain("Saltoro Kangri", 25409, pakistan).
```

In Chapter 4, predicates with only single arguments were used: an arity of one. But, while coding in Prolog, you will find that most programs depend heavily on predicates with much greater arities. For example, you might want to use the fact

```
eats (eric, shark).
```

to describe the relationship (predicate) "eats" between the two objects (arguments) "eric" and "shark". This could easily be translated to mean "Eric eats shark." Remember, though, that the meaning of a predicate is what the programmer defines it to be. This same predicate could just as easily mean "A shark eats Eric!" It is up to you to decide which way a relationship works with its arguments, then to stick to that form of the relationship throughout your program.

Comments can be written in your program to show how you intend a predicate to relate to its arguments. However, the comment will only help the next reader of the program to understand the meaning of the predicate; it will not help the Prolog compiler understand your intention. If, in a program, you mix up the way a relationship interacts with its arguments, you will not get a compile error, but you will get unexpected results when your program is run.

When programming, you may wish to use a predicate of sizable arity to describe an object. For instance, the following eight arity predicates should handle most of the basic, descriptive characteristics of a computer system:

```
computer (ibm_pc, ram_640K, two_floppy, dos_3, two_serial_ports,  
one_parallel_port, ega, multisync)
```

As you can see, a predicate can hold as many terms as is necessary to describe the relationship at hand. The larger the arity, the more descriptive the relationship will be. But keep in mind that there are only so many objects a relationship can have before the predicate becomes unwieldy. Predicates can also be written with no arguments at all, but that's a rare case and their use is rather limited.

Turbo Prolog 2.0 added the ability to declare a single predicate more than once, using a different arity for each declaration. While this is a nice feature, notice that these predicates are really separate predicates altogether. This concept is made clearer in the discussion of how Prolog's matching mechanism works.

Goals with Multiple Arguments

After creating a short program consisting of single-argument predicates in Chapter 4, you interrogated that program with single-argument goals. Programs that employ predicates with greater arities work with more complex goals. Here are a few clauses constructed to provide a simple example:

```
watches (bill,bob).           /* bill watches bob */  
watches (john,jane).  
watches (fred,felicia).  
watches (mitch,bill).  
watches (louise,greg).  
watches (bob,bob).  
watches (fred,greg).  
watches (bill,jon).
```

The predicate “watches” means that the object in the first position watches or observes the object in the second position. This is shown to the reader of the program by a short comment placed after the first `watches()` predicate. When typing in these facts, remember to end each clause with a period.

Add the predicate and domain declarations, along with a title line to identify the program, as shown in Figure 5-1. Be sure to save the program when you are finished with it.

Notice that all of the arguments to the predicate `watches()` contain the same domain, “person”, which is declared to be of type “symbol”. This shows that each argument supplied to `watches()` must belong to the domain “person” and that the value in the arguments must follow the rules that define the type symbol.

This declaring of argument types is unique to Turbo Prolog. While typing arguments may seem like a given to Pascal programmers, typing is unnatural to veteran Prolog programmers. Keep in mind that Prolog was formed out of the concepts of formal logic, and that Prolog was originally intended to solve logistical problems.

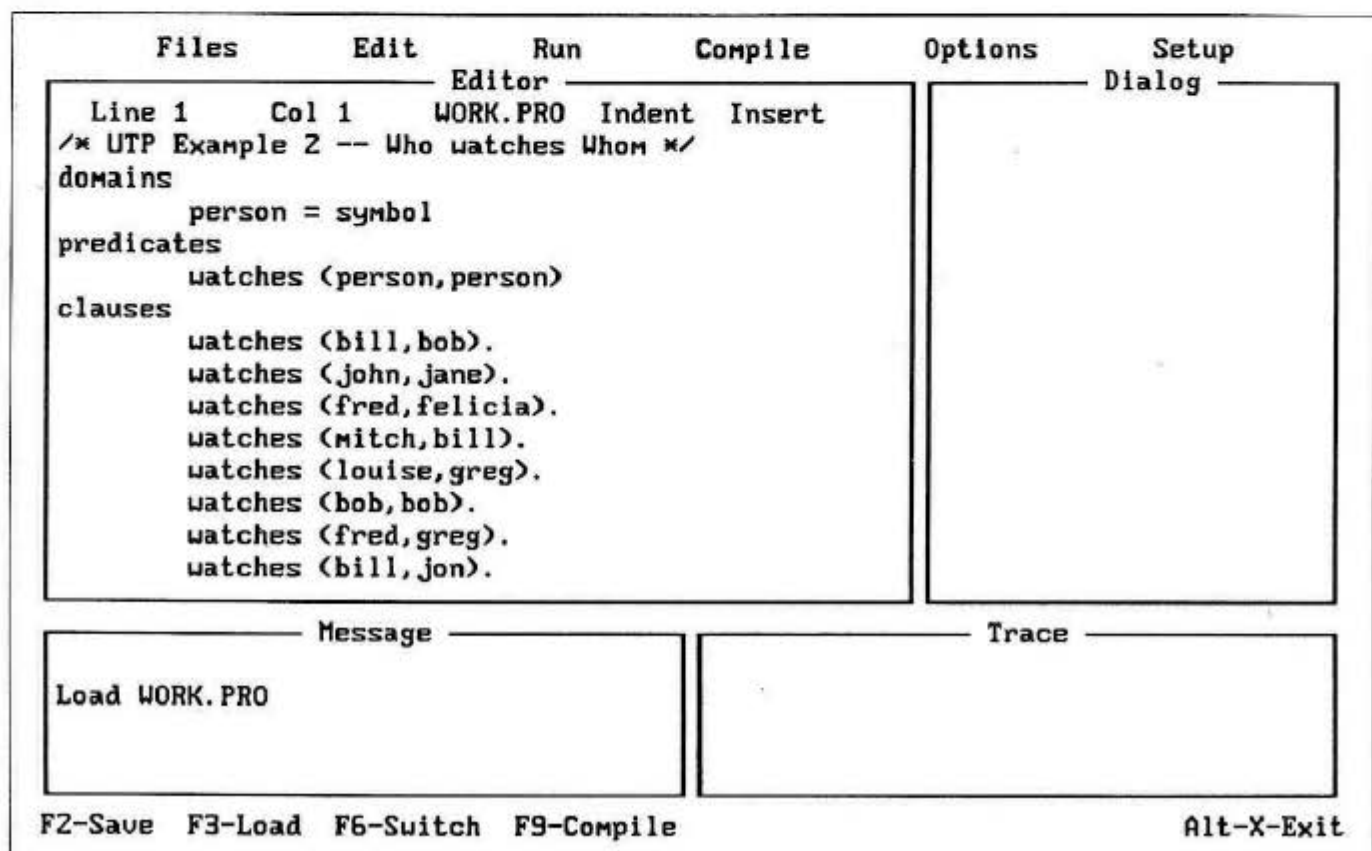


Figure 5-1. Example program number 2

Normally, little thought is given to how much memory it will take to reason out a certain logistical problem. The nature of the problems that Prolog was designed to solve dictated that large, mainframe computers be used to house the Prolog interpreters. When Prolog was introduced to the personal computer, the interpreters were slow and memory-intensive. Adding the ability to declare argument types to the Prolog language has allowed the speed to increase while keeping the search space needed to solve a problem to a minimum. With the type declarations introduced with Turbo Prolog, a wide range of problems can now be quickly and effectively solved using the Prolog language.

Prolog's Matching Mechanism

Compile and run the program in Figure 5-1 with the ALT-R hot-key command. Now pose some external queries (or goals) for the program to solve. When queried, Turbo Prolog will search from the top of the program to the bottom, looking through the list of facts for one that matches the goal you have presented. Prolog will first try to match the predicate name, then check to see if the fact has the same arity as the goal. If a match is found, Prolog will then try to match the first argument of the predicate with the first argument of the goal. If successful, Prolog will continue in its attempt to match the goal by trying to match the second arguments, if any exist. Eventually, if a full match is found, Prolog will come back to the Dialog window with the answer "yes." If it doesn't find a match, Prolog will return with "no" as the answer to the query. For instance, try the following goal and watch what Prolog comes up with:

watches (fred,greg).

Turbo Prolog quickly returns "yes" to the Dialog window, indicating that it has found a match to the given call.

Using Variables in Goals

If one of the arguments in the goal happens to be a variable, Turbo Prolog will work through the clauses, again attempting to find a match to the goal by first matching the predicate name and then the predicate arity. After a match is found, the matching process will continue by matching the first argument in the goal with the first argument in the matching predicate. The difference appears when the argument being matched in the call (or goal) happens to be a *free* (not bound) variable.

Since a free variable will match with any Prolog term, it will “instantiate” with any respective argument in the matching predicate. When a match to a query is found with a fact, Prolog instantly brings back the value it has found by binding the free variable in the call to the value found in the match. When a query is provided from the Dialog window with free variables, Turbo Prolog instantly reports the findings of these variables back to the Dialog window. For example, try the following goal, which contains a variable:

```
watches (fred, Who).
```

This brings the response

```
Who = felicia  
Who = greg  
2 Solutions
```

You can use a variable in any argument position you wish in order to return the value or values you desire. For example, use variables in both argument places with a query to the above program. Try the goal

```
watches (Who,Whom).
```

This call returns a complete list of all the values from the watches() clauses in the program, as shown in Figure 5-2. This is because

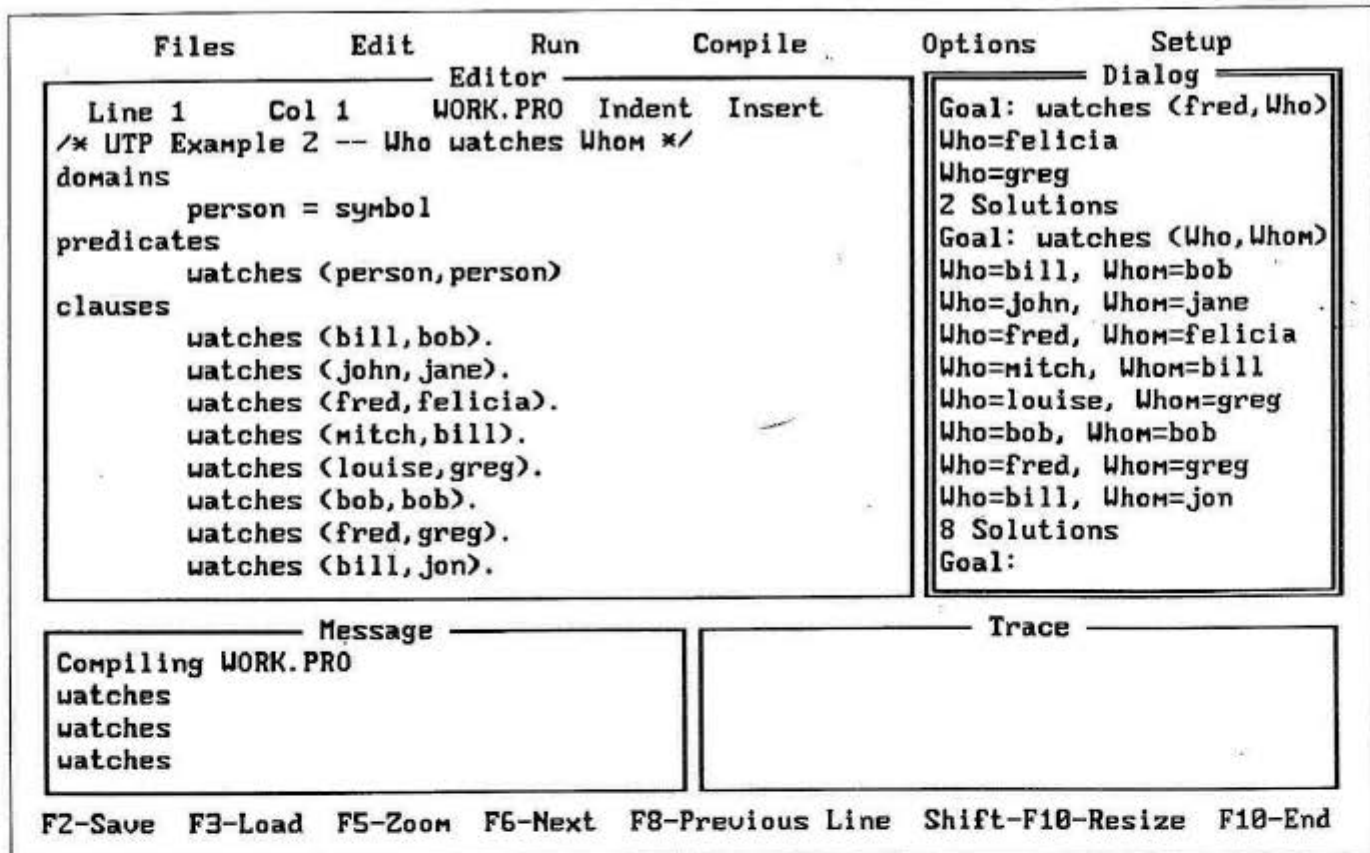


Figure 5-2. Using two variables in an external goal

Turbo Prolog will find all solutions to the variables presented in queries entered at the Goal: prompt. With two free variables in the query, the call will match all `watches()` facts in the program, and all the values found will be returned and reported in the Dialog window. Incidentally, if you want to repeat a goal without retyping it, press F8. This shortcut command can be used with any goal that has been entered in the Dialog window.

Remember that while a bound variable carries an exact term or value, a free variable can match with any term or value. A free variable in a call can even match with another free variable. However, once a free variable in the calling predicate finds a match in the matching predicate, the variable in the call becomes bound to the value found. The variable will then carry this value throughout the rest of the clause (or goal) being processed. Once the goal is fully satisfied, all variables will release the values they are carrying. The carrying and releasing values may at first seem a little awkward, but the concept will become clearer as you learn how Prolog backtracks to find its solutions.

The Anonymous Variable

Another type of variable that is handy in some situations is the *anonymous* or *blank* variable. It is written as a single underscore character—no letters or digits. The anonymous variable is used in the same places that standard variables are used; the difference is that the anonymous variable never returns a value. In other words, the anonymous variable can hold the place of an argument whose value you do not need to know about.

Suppose, using the above example program, that you want to know if Louise has watched anyone, but you do not care who that anyone is. You could pose a query using the anonymous variable

```
watches (louise, _).
```

After finding a match to the goal, “yes” will be returned as the response instead of an indication of the number of solutions. And because the anonymous variable was used, you are not told who Louise watches.

The anonymous variable can also be used when representing facts in the Clauses sections of your programs. Add the following fact to your program:

```
watches (_,louise).
```

You have now stated that “everyone watches Louise” (or that “anyone watches Louise”) is a true fact. Because the anonymous variable can find a match with any Prolog term, the above fact is interpreted as “anything” watches Louise. This can be shown with the goals that follow.

```
watches (claudio,louise).  
watches (micki,louise).
```

Notice that both of these queries will return “yes,” even though the names Claudio and Micki are not entered in the database.

Here, you’ve queried a very small database for a simple piece of information. With a huge database, Prolog would be able to look

up this particular information much faster than anyone would ever be able to do by hand. Combined with rules and other logical constructions (that are explained along the way), Prolog can far outstrip any manual or nonautomated scheme for searching a set of facts.

Compound Goals and Rules

Goals can be more complex than those shown so far; a single goal can call on several facts in order to be proven true. For example, you may wish to know, "Is it true that Louise watches Eric and that Eric watches John?" To link the parts of a query such as this requires the use of the logical statement AND.

Goals Containing AND

If you pose the query

`watches (bill,bob) AND watches (john,jane).`

you'll be asking, "Can we prove it to be true from the known facts that Bill watches Bob and that John watches Jane?" This compound goal will be judged as true only if both of the individual goals are proven true.

In a multiple-part goal connected with ANDs, Turbo Prolog will begin with an attempt to solve the first subgoal, just as if it were a single goal. If a match cannot be found with the first call, the entire goal will fail. On the other hand, if the first call succeeds, Prolog will then attempt to match the second subgoal, the part of the goal after the AND statement. Only if all ANDed subgoals are found to be true will the entire compound goal succeed.

Most implementations of Prolog use a comma (,) to represent the AND statement and Turbo Prolog is no different in this respect. Because of this, the compound goal shown above could be written as

```
watches (bill,bob), watches (john,jane).
```

Within compound goals, you can use variables and anonymous variables, just as you would in simple goals. Pose the query

```
watches (Who,jon), watches (_,Who).
```

and you'll get the result

```
Who = bill
```

```
1 Solution
```

The query asks, "Is there someone (Who) who watches Jon, and is that same person (Who) also watched by someone (anything) else?" Judging from the response, "Who" turns out to be Bill. Later on, you will be shown how to use the trace facility to figure out what logic Prolog used to come to this conclusion.

Goals Containing OR

Compound goals using OR can return true after only a single set of subgoals has proven to be true. For example, while the compound goal

```
watches (bill,bob) AND watches (john,jane).
```

returns true only when both "bill watches bob" and "john watches jane" are proven to be true, the goal

```
watches (bill,bob) OR watches (john,jane).
```


returns true if either "bill watches bob" or "john watches jane" are found to be true. Turbo Prolog allows a semicolon (;) to replace OR.

Notice how the same query can be posed as two entirely separate queries:

```
watches (bill,bob).  
OR  
watches (john,jane).
```

If either case returns true, you know you have a solution.

Extreme care should be taken when using the OR statement in your programs. Unexpected results can turn up because of the way in which Prolog processes goals containing ORs. Use the OR statement (the disjunction) in your clauses only when you have a firm grasp of how Prolog backtracks in its search for solutions.

Adding Rules to Your Program

Facts aren't the only inhabitants of Clauses sections. Prolog rules can also be entered into your information set. A *rule* is a relationship that is proven to be true after a separate set of relationships has been proven true. An example of a rule can be expressed with the statement, "Eric likes to eat fish if the fish is broiled." To say that Eric likes to eat a particular fish, you must first prove that the fish in question has been broiled. The corresponding rule in Prolog looks like this:

```
eats(eric,Fish) IF broiled(Fish).
```

Whereas a fact is stated as a predicate followed by its arguments, a rule has three parts to it. The *head* of a rule is identical to the facts you are used to. In the example rule, "eats(eric,Fish)" is the head of the rule. Notice that the head of a rule can only hold a single predicate and is followed by the IF symbol instead of a period.

Following the second part of the rule (the IF symbol) is the *body* of the rule. The body is made up of one or more *subgoals* and is identical to the queries that you have been entering at the Goal: prompt. The list of subgoals in the body of a rule is connected by AND or OR statements, and may be listed on separate lines to make for easy reading, like this:

```
nervous (Who) IF  
    watches (bill,Who).
```

The IF statement may be represented in Turbo Prolog by a colon followed by a hyphen (:-). Thus, the rule in our example can be written as

```
nervous (Who) :-  
    watches (bill,Who).
```

For the remainder of the book, the standard symbols for AND, OR, and IF will be used. Since most Prolog programs are written with these symbols, it is important to get used to them now while you are learning to code in Prolog.

All facts and rules in Prolog must end with a period. Oddly, facts and rules are the only types of code statements you will use in Prolog programs. Each rule or fact is known as a Prolog *clause* and, put together, the clauses in your program make up the program's *database*. Unusual as it may sound, a Prolog program is actually made up of the data that the program operates on. This is entirely different from other programming languages, such as C and Pascal, where the code you write acts on a set of data that is kept separate from the program code itself.

Unlike a fact (which is inherently true and lets a call return instantly when matched), a rule must go through the process of proving its subgoals before a return can be made. Rules take Prolog beyond the status of a searchable dictionary and into the realm of a logical, "thinking" machine capable of deducing relationships from the facts that are known to be true.

A New Set of Goals

When Prolog is going through a search, trying to find a match to a query, it can check for a match with either a fact or the head of a rule. If a match is found with a rule, that rule is said *to fire*. If the rule concerning fish and Eric were in the Clauses section of your program, and you entered the query "eats (eric, What)", a match could be found with this rule. Once a rule fires, the list of subgoals contained in that rule must be proven true before Prolog can continue with its current processing. The list of subgoals in the rule is placed ahead of the goals currently being processed and these new subgoals become number one priority.

In the rule above, the list of subgoals is rather short. All that needs to be done for this rule to succeed is to find a match for the single subgoal "broiled(Fish)." Once this is done, Prolog can return to the call made to the rule.

Let's add the following rules to the end of the example program (be sure to put periods at the end of all your rules):

```
happy (greg) IF watches (louise,greg).
happy (frank) :- watches (louise,frank).
nervous (Who) :- watches (bill,Who).
```

After adding these rules to your program, you will have to add some new predicate declarations before you can compile the expanded program. Type the predicate declarations shown in Figure 5-3. Notice that you don't need to declare any new domains, since all the new predicates use a domain that has already been declared.

After compiling the program with the added rules, try the goal

```
happy (greg).
```

Just as before, when trying to match the goal with a clause, Prolog starts at the top and works its way to the bottom of the clauses, moving from left to right on each clause in search of a match. Prolog will search through and past the watches() facts and on to the clauses defining the predicate happy(). When it finds the first happy() predicate, which has "greg" as an argument, Prolog knows it has hit possible paydirt.

Files	Edit	Run	Compile	Options	Setup
Line 1	Col 1	WORK.PRO	Indent	Insert	
/* UTP Example 2 -- Who watches Whom */					
domains					
person = symbol					
predicates					
watches (person, person)					
happy (person)					
nervous (person)					
clauses					
watches (bill, bob).					
watches (john, jane).					
watches (fred, felicia).					
watches (mitch, bill).					
watches (louise, greg).					
watches (bob, bob).					
watches (fred, greg).					
watches (bill, jon).					
happy (greg) if watches (louise, greg).					
happy (frank) if watches (louise, frank).					
nervous (Who) if watches (bill, Who).					
F1-Help F2-Save F3-Load F5-Zoom F6-Next F7-Xcopy F8-Xedit F9-Compile F10-Menu					

Figure 5-3. Adding rules to the example program

The rule now fires, and the next step is to see if the body of the rule can be satisfied. The list of subgoals in the body now becomes the new, temporary set of goals that must be satisfied before the program can move on.

After Prolog calls the first subgoal in the `happy()` rule, the search will go back to the top of the clauses and try to find a match to the call `"watches(louise, greg)"`. When it finds a match, the body of the rule is satisfied and Prolog can return to the head of the rule, judging the entire rule to be true. A return can then be made to the Dialog window, which proclaims that the query is true.

By the way, you may have noticed that the last new rule added to the program makes use of variables in both the head and the body. Variables are permitted in rules, as are anonymous variables. The usefulness of including variables in rules is explained in the next section, when backtracking is described.

If, while searching for a solution to a call, a match is found with the head of a rule, all the subgoals in the body of that rule must be satisfied before any values can be returned to the call. Once a rule's subgoals have all been proven true, the rule is said to *return*. At this point, all the values in the head of the rule are passed back to the arguments in the goal that was originally called.

If a goal has any free variables when the call is first made, these variables will become bound to the respective values in the rule's head when the return is made.

After a return is made from a rule, all the variables in that rule will be freed of their bindings. It is important to see that in Prolog, the scope of a variable is one clause. Once the end of a clause is reached and a return is made, the variables in that clause will free the value bindings they have taken on.

There is no regulation restricting the number of subgoals a rule may have. In fact, a rule can have many subgoals, such as this:

```
happy (freda) :-
    happy (eric) ,
    happy (louise) ;
    watches (bill,bob) ,
    nervous (—).
```

This rule, depending on the way the program is written, might state that, "Freda is happy if Eric is happy and Louise is happy, or Freda is happy if Bill watches Bob and something (anything) is nervous." This may not make much sense, but it's an example of how a rule can be written.

The Search for Solutions

It has previously been established that Prolog will try to find all the solutions to a given query. This is possible because of the backtracking mechanism contained in the Prolog language. Backtracking is also the key element to the way in which Prolog goes about its search for solutions. When processing a set of subgoals, Prolog attempts to find match after match. The word to watch here is "attempt." What happens if Prolog doesn't find a match? What happens if the attempt at matching fails?

Up until now, it has been assumed that everything will gracefully succeed, returning "yes" to all your queries. When a call is made that cannot be proven to be true, the call is said to *Fail*. Failure can also happen when Prolog attempts to satisfy the body of a rule because a subgoal in the body can fail.

The Backtracking Bread Crumbs

It is this failing that kicks in the Prolog backtracking mechanism. When a call fails, Prolog can no longer consider its current path to be one that will lead to the eventual solution of the problem. Prolog must back up and continue its search somewhere else to find a successful outcome. This backing up to find an alternate path is called *backtracking*.

The key to backtracking is the bread crumbs that Prolog leaves behind as it hikes through its searches. When Prolog comes across a *fork* in its search path, it places a "bread crumb" there to mark the crossroad. Every time Prolog comes to a fork and there is at least one path that must be left untried, a bread crumb is left behind. If something goes wrong, Prolog simply backs up to the last bread crumb, and resumes its search down the untried path. When Prolog backs up to a bread crumb, and there is only one path left untried, Prolog picks up the bread crumb as it begins down the last untried path at this fork. Although this description may sound like a fairy tale, it is essentially how Prolog works! Here is an example showing backtracking in action.

```
/* Mow the Lawn Backtracking Example */

Predicates
  lawn_needs_mowing
  grass_mowed ( symbol )
  sun_shining ( symbol )
  raining ( symbol )
  day ( string )

Clauses
  lawn_needs_mowing IF
    sun_shining(true).
  lawn_needs_mowing :-
    raining(false) ,
    grass_mowed(false).
  lawn_needs_mowing :-
    day("Sunday").

  sun_shining(false).

  grass_mowed(true).

  raining(false).
  raining(maybe).

  day("Sunday").

/* END */
```

Given this long and complex program, let's unravel the mysteries of backtracking. By compiling and running this program, you can

find out many interesting things. You can find out if the sun is shining by applying the following goal:

```
sun__shining(ls__lt).
```

Or you can query to find out what day of the week it is (showing that your computer knows only what you tell it). But the real fun comes when you want to find out if it's time to mow the lawn. Typing the goal

```
lawn__needs__mowing.
```

sets Prolog in search of a solution. When trying to find a match to this call, Prolog looks at the first clause in the program and promptly decides that it is on the right track. However, since Prolog sees that there are other possible paths it can take (the other rules defining the predicate `lawn__needs__mowing()`), Prolog sets one of its bread crumbs down for future reference. At this point, the first rule fires, and the subgoals in this rule now become the number one priority.

Prolog can only conclude that the rule

```
lawn__needs__mowing :-  
    sun__shining(true).
```

is true if it can prove that the subgoal

```
sun__shining(true).
```

is true. A search now begins for the solution to this new subgoal. Since no corresponding clause can be found to match the call, the subgoal fails. Here, Prolog calls on backtracking to help solve the puzzle. The search backs up to the last bread crumb and continues its search from there. The next path to try is the second clause defining the predicate `lawn__needs__mowing()`. After taking this path, there is still one more path left untried, so another bread crumb is left by the trail marking the last clause defining `lawn__needs__mowing()`.

Now, the second `lawn__needs__mowing()` rule fires:

```
lawn__needs__mowing :-  
    raining(false) ,  
    grass__mowed(false).
```

This rule may seem slightly more complex due to the number of subgoals in the body. Not to worry, though, because processing continues by giving these subgoals the highest priority. A call to “`raining(false)`” is made, and quickly returns successful. But notice that in doing so, Prolog passed another path that might have led to a solution: the second clause defining the predicate `raining()`. Because of this, a bread crumb is placed next to the first `raining()` predicate when this clause is tried. The second subgoal in the rule can now be tried, but it is your lucky day — the grass has already been mowed (see the clause defining the predicate `grass__mowed()`), and the call to “`grass__mowed(false)`” fails.

Since another call has failed, backtracking backs you up to where the last bread crumb was placed and Prolog tries to resolve the call made at that point. The process of attempting to resolve a call is known in Prolog as *REDOing* a call. Here, Prolog REDOes the call to

```
raining(false)
```

this time removing the bread crumb as Prolog tries the last clause defining the predicate `raining()`. (Remember, when no untried paths are left for a particular call, no bread crumbs should be left to mark another possible solution to that call.) The call cannot be proven true, so Prolog again backs up to the last crumb placed, which now happens to be next to the second rule defining the predicate `lawn__needs__mowing()`. Since Prolog will now be trying the last path to this call,

```
lawn__needs__mowing :-  
    day("Sunday").
```

it again picks up the bread crumb as it goes by. Processing continues, REDOing the call to `lawn__needs__mowing()` for the last time. Unfortunately, your good luck has run out, and you see that the proposed goal returns a “yes.” Off you go to mow the lawn!

It is important to note that when Prolog backtracks past a point

at which a variable is set to a value (the point at which a variable originally becomes bound to a value), the variable is then freed of the value it had taken on. During the processing of a clause, backtracking is the only way that a variable can be freed of a value once it is bound. And again, once a clause succeeds, all the variables in that clause are then set free from their bindings.

Tracing

Backtracking may be difficult to describe, but it is easy to see in action when you use Turbo Prolog's tracing features. Tracing lets you take a peek at the step-by-step processing of your program, instead of only seeing the results pop up. Tracing makes the development environment stop the program at each step of the way to tell you what it has done, where it did it, and what the variable bindings are at that point. Watching what a program does as it moves and backtracks its way through a program can be accomplished by adding the compiler directive

trace

to the top of your program. To illustrate how the trace compiler directive works, you will use the "watches" program written previously. Figure 5-4 shows the addition of the trace compiler directive. Adding the trace directive tells the compiler to run the program in Trace mode. If you then compile and run the program, you'll see the normal

Goal:

line appear in the Dialog window. But watch what happens in the other windows as you enter the simple goal

happy (Who).

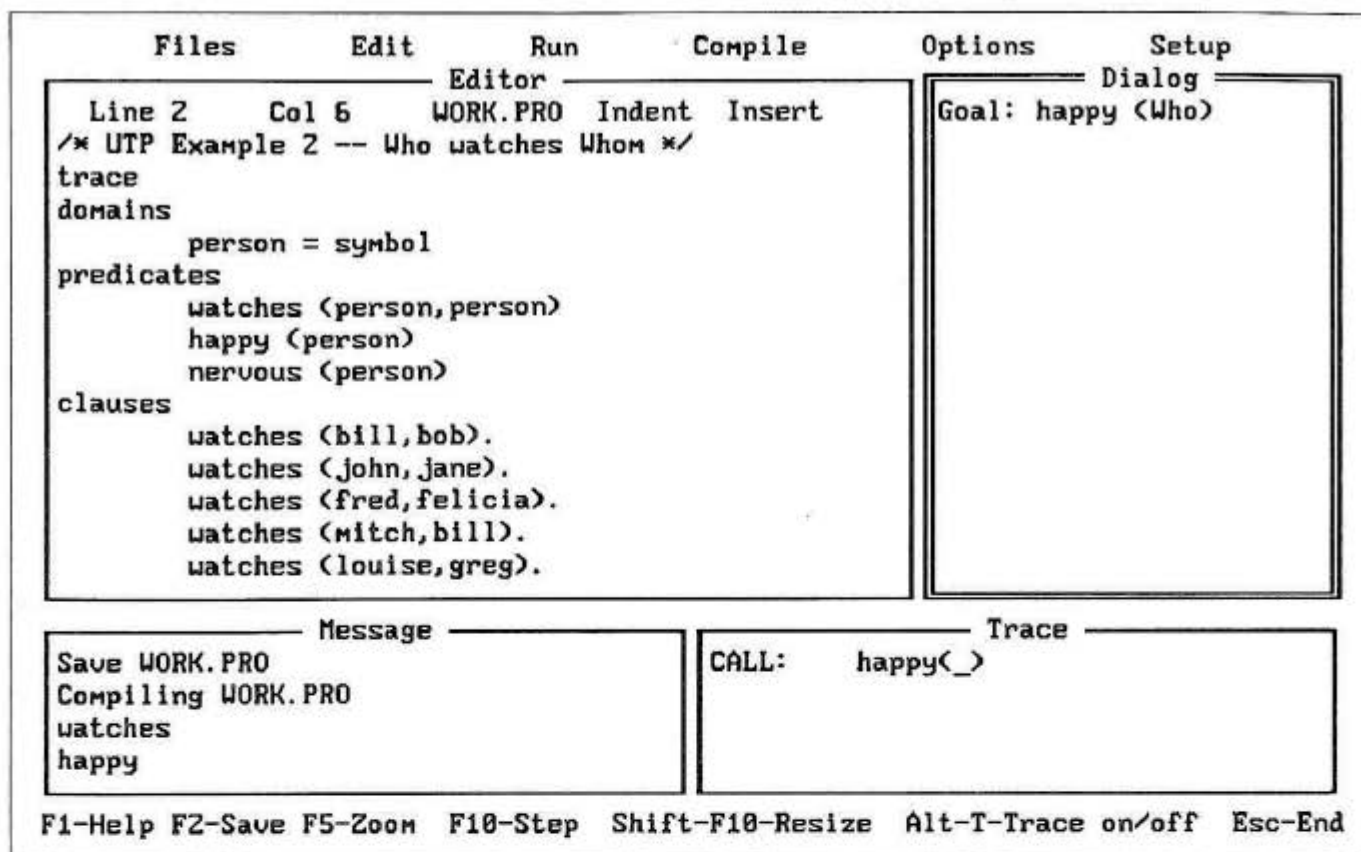


Figure 5-4. The first step in tracing the example program's progress

In the Message window you will see that the program is compiling. The Dialog window places the cursor at the beginning of the goal you have typed, and the Trace window tells you that Prolog is "CALLing" the

happy (—)

subgoal. The first step Prolog takes is to find a match with the call to happy(), a predicate with a single argument.

Now move on to the next step by pressing the F10 key. This is the single-stepping key for the rest of the trace. After a single press, you'll see the cursor land on the first happy() predicate in the clauses section of the Editor window, which shows that Prolog has found a match with both the predicate name and the predicate arity. Press F10 again and you'll see the cursor move to the body of that rule, showing that a true match has been found. In the Trace window you'll see that Prolog is now "CALLing" the new subgoal:

CALL: watches ("louise","greg")

Another press and you'll see Prolog move back to the beginning of the clauses to start its matching process for this subgoal. After it fails to match the new goal with the first clause, it will "REDO" the matching attempt with the next possibility, and you'll see the Trace window show

REDO: watches ("louise","greg")

Keep pressing F10 and Prolog will eventually come to a clause that matches the new goal. When it does, it will show

RETURN: watches ("louise","greg")

in the Trace window, and will then skip back to the rule it was attempting to solve. The return will show as

RETURN: *happy("greg")

in the Trace window, since its body was successfully matched. The asterisk following "RETURN" means that Prolog has left one of its backtracking bread crumbs at this point. If, later on, a call fails, Prolog knows it can return to this point in order to REDO the call and look for an alternate solution. The next step will return "greg" as a solution to the Dialog window. Turbo Prolog will immediately "REDO" the subgoal

REDO: happy(—)

in an attempt to find all the solutions to the original goal. By using a variable in the goal, you have left Prolog open to find more than just a match that returns a "yes." The variable will return a value to every solution that it can find. Because of this, once a solution is found, the original goal is immediately redone in an attempt to find more solutions.

This time, Prolog works its way to the clause

happy (frank) :- watches (louise,frank).

and makes the body of this rule the new set of subgoals. After working through all the clauses with this new subgoal, this call will fail, and Prolog will tell you so in the Trace window with

```
FAIL: watches ("louise","frank")
```

The entire rule has thus failed to find any more solutions, and the program will finish by pointing out (in the Dialog window) the one solution it has found. Turbo Prolog will now be ready to process the next goal you wish to give.

Compound Goal Tracing

Watching a compound goal work through a program is not much different than tracing a simple goal. When Prolog processes the goal

```
watches (Who,jon), watches (_,Who).
```

as shown earlier in this chapter, it first looks for a `watches()` predicate with an arity of two. If it finds one, and "jon" is the value of the second argument, it binds the variable "Who" to the respective value it finds in the matching predicate. Next, Prolog starts at the beginning of the clauses, looking for a `watches()` predicate, again with an arity of two and with a second argument equal to the value that is bound to the variable "Who".

Since the first argument of the second subgoal is an anonymous variable, this argument can match any value or term. If a match is found, Prolog can return that the goal is true by reporting the findings of the variable "Who".

Again, since the goal contains a variable, once one solution is found the original goal is redone in an attempt to find all the solutions. Try tracing this goal as an exercise to help you understand how Prolog backtracks to find its solutions.

As another exercise in tracing, try typing the "mow the lawn"

program and run it in Trace mode. Try following the CALLs, RETURNs, FAILs, and REDOs to see how Prolog backtracks to find solutions to the queries you pose.

Not only is tracing a powerful debugging tool, it is also a powerful aid in understanding how Prolog finds its solutions. If a program comes up with unexpected results, or if you are mystified about how a particular predicate works, put the program into Trace mode and watch how Prolog arrives at its answers. When more complex predicates are introduced later on in the book, the Trace mode may be your ticket to fully understanding how Prolog has processed those predicates.

Using Not()

The predicate `not()` changes the truth of a subgoal from its original state (true or false) to its opposite state (false or true). If you try the simple goal

```
watches (bill,jon).
```

you get a “yes” response, as can be expected. But, if you try the goal

```
not (watches (bill,jon)).
```

you’ll get a “no” response. (The extra set of parentheses is necessary for the logical mind of Prolog to know that you’re applying “not” to everything within the following set of brackets.) On the other hand, a goal that would normally fail will return true when tagged with “not”. The goal

```
watches (bill,felicia).
```

will fail because there was no fact to match the call in the clauses section. Yet the goal

```
not (watches (bill,felicia)).
```

will yield a “yes.” This is because you’re asking the opposite of the original question, in other words, “Is it not true that Bill watches Felicia?” Yes, it is not true, because it cannot be proven that Bill watches Felicia. Note that when using the `not()` predicate, all variables inside `not()` must be bound before `not()` is called. If any free variables exist when `not()` is called, your program will end with a runtime error, causing problems for all users.

Pressing Onward!

This chapter covers just about all you will need to know to program in Prolog. However, many of the sections are covered quickly, not leaving much time for you to digest the material presented. This chapter is intended to be a reference guide to how Prolog works. Once you have become comfortable with using Turbo Prolog, rereading sections of this chapter will help to clear up many of the questions that will undoubtedly crop up concerning backtracking, the binding of variables, and the matching of calls.

6 The Sections of a Turbo Prolog Program

If you are used to making declarations in a language such as Turbo Pascal, you'll skim through this chapter pretty quickly because declaring certain aspects of your program will be nothing new. But if you have been programming in another implementation of Prolog, making declarations will be a new experience in Prolog programming.

Turbo Prolog requires that you declare the different domains and relationships you use in your program. Although Turbo Prolog includes a wealth of built-in relationships, creating customized programs requires that you define and declare your own relationships and the "domains" of the objects used by them.

A Turbo Prolog program is made up of different *program sections*. Inside these sections, you will declare and write the predicates that make up your program. You should already be familiar with the three most basic sections: the Clauses section, the Domains section, and the Predicates section. This chapter outlines the sections that a Turbo Prolog program is made up of and explains the benefits of using each one.

Clauses Section

Since most of this book deals with the code you write in the Clauses section, it is only mentioned here for the sake of completeness. The Clauses section is where the actual program code takes residence. In the other sections of your program you declare the pieces of the program you are going to write. In the Clauses section, you write (or define) the facts and rules that you have declared. This section is considered the heart of the program because it is here that the processing takes place.

Predicates Section

Two types of predicates are used in Prolog programs: *system* (or *built-in*) *predicates* and *user-defined predicates*. System predicates do not need to be declared because they are a built-in part of the language. User-defined predicates, on the other hand, are first declared in the Predicates section, then written in the Clauses section.

User-defined Predicates

Any predicate that you define must first be declared in the Predicates section of your program. A *predicate declaration* consists of the predicate name followed by its argument declarations. The following example shows the format used for declaring predicates:

```
predicatename(argument1,argument2,...,argumentN)
```

A predicate can have zero or more arguments in its declaration. If a predicate is declared to take one or more arguments (as most do), each argument must be given a specific domain in the declara-

tion. These argument declarations reflect the data types that the predicate uses and will either be a standard domain type or a user-defined domain. (If the idea of domains and data types seems foreign to you, don't worry. Domains and types are covered in the following section.)

While a predicate name can consist of any combination of letters (both upper- and lowercase), digits, and underscores, the first character must be either a lowercase letter or an underscore. Predicate names can be up to 250 characters in length and must not contain any spaces. Minus signs, asterisks, and slashes are also not allowed in predicate names.

Multiple Predicate Declarations

A single predicate name may have more than one declaration in the Predicates section. Multiple predicate declarations come in handy when a predicate needs to handle more than one specific data type in a given argument position. Also, multiple declarations allow a predicate name to be defined using a different number of arities (remember, the arity of a predicate is the number of arguments it takes). The following predicate declarations are perfectly legal in Turbo Prolog:

```
predicatename (argument1, argument2)  
predicatename (argument3, argument4, argument5)
```

or

```
has__toes (person,number)  
has__toes (animal,number)
```

With these declarations you'll be able to use either predicate in one of two ways. A call to `predicatename()` can be made with either two or three arguments, making sure that the arguments match the types declared. With the use of multiple declarations, the predicate `has__toes()` can be called with one of two different sets of data types.

System-Defined Predicates

Using the predicates that Turbo Prolog provides is as easy as making a call to the built-in predicate needed. Just be sure that the correct number of arguments is used and that the values supplied in the call match the data types the predicate expects to see. Below are examples of how to use some predicates provided by the system.

Making Windows and Writing To Them Programs written in Turbo Prolog can be made to have polished *user interfaces*, where the user of a program is given a complete environment to work with. Part of this interface can be accomplished with the windowing capabilities that are included with Turbo Prolog.

Version 2.0 of Turbo Prolog added a new window-making predicate to the one that existed in version 1.1. The use of these two predicates is basically identical. A call to the built-in predicate `makewindow()` will make a window (much like the windows in the Turbo Prolog environment) with the dimensions and location of your choice. Along with this, you can declare the window to be either framed or unframed and choose the colors of both frame and window. Here is an example showing a call to a basic version of `makewindow()`:

```
makewindow(1, 5, 2, " Window 1 ", 0, 0, 25, 80)
```

With this call a window is made on the computer screen. According to the arguments given, the window is number 1 and the color of the text inside the window has an attribute of 5 (see Table 6-1 for a description of the color attributes). The third argument to `makewindow()` indicates whether or not the window has a frame. If the argument given has a value of zero, no frame is made. In this call, the frame has an attribute of 2, which also relates to the colors shown in Table 6-1. When making windows, you have the option of including a message in the window. The call above indicates that "Window 1" will be displayed at the top of the frame.

To choose a color attribute, pick the background color you want, then add the foreground color of your choice. If you want the characters to blink, add 128 to the attribute value. For example, to have a red background with green characters, the attribute would equal $64 + 2 = 66$. And to have the characters blink, the attribute

would equal $66 + 128 = 194$. If you have a monochrome monitor, use an attribute of 7 for best results.

The last four arguments to `makewindow()` give the location and dimensions of the window. The first two arguments indicate where the upper-lefthand corner of the window will be; according to the arguments 0,0, the window starts in the upper lefthand corner of the computer screen. The next argument in this series determines how many rows on the screen the window will take up. In this case, the window made is 25 rows deep. The last argument details the number of columns across the screen that the window will occupy;

Attribute	Value	Foreground Colors
	0	Black
	1	Blue
	2	Green
	3	Cyan
	4	Red
	5	Magenta
	6	Brown
	7	White
	8	Gray
	9	Light blue
	10	Light green
	11	Light cyan
	12	Light red
	13	Light magenta
	14	Yellow
	15	High-intensity white

Attribute	Value	Background Color
	0	Black
	16	Blue
	32	Green
	48	Cyan
	64	Red
	80	Magenta
	96	Brown
	112	White

Table 6-1. Color Attributes

here, the window is 80 columns wide. If your computer is in normal Text mode, a full-sized window will be made on the screen.

Whenever a window is created, that window then becomes the active window. All input and output to the computer screen takes place through the active window (for more on input and output, see Chapter 10). When a window is made, the cursor is automatically positioned at the upper-lefthand corner of the window; this position is always position 0,0, indicating row 0 and column 0 of the new window.

It is not enough to simply make a window. For it to be useful, you must place something in the window you have just made. Turbo Prolog provides several ways to output text, the simplest being with the built-in predicate `write()`. This predicate can take dozens of arguments, and the result of the call will write the values given to the currently active window. For example, when the following call to `write()` is made, the sentence "Hello out there!" is output to the active window:

```
write("Hello out there!")
```

Often you will wish to start a sentence on a new line, so Turbo Prolog offers this built-in predicate:

```
nl
```

The predicate `nl()` doesn't use any arguments. When called, it moves the cursor down one line on your computer screen.

The following short program demonstrates the power of making windows and writing to them. Use the goal "run" to start the program.

Predicates

`run`

Clauses

`run :-`

```
makewindow(1,2,3," Window 1 ", 0,0,25,40) ,
X = "Hello, there!" ,
Y = "It's been nice." ,
write(X), nl ,
makewindow(3,2,1," Window 2 ",0,40,25,40) ,
write("From Window 2: "), nl ,
write(Y, " Goodbye!") ,
makewindow(3,6,1, " Window 3 ", 10,10,10,60) ,
write("Press any key..."), nl ,
readchar(_).                % wait for a key to be pressed
```


Most of the built-in predicates that you will need are introduced in this book. A full listing of all the Turbo Prolog predicates is provided in the appendix. Also, an entire section in the *Turbo Prolog Reference Guide* is devoted to the definition and use of all the built-in predicates included with the language. Along with each predicate description is an example program showing how it can be used. Take time to read through this section of the Turbo Prolog guide to see the features of the language that are available to you.

Predicate Flow Patterns

You may notice while browsing through your *Turbo Prolog Reference Guide* that each predicate is given a flow pattern (or a set of flow patterns). The syntax used to show a flow pattern is an open parenthesis followed by a series of i's and o's (separated by commas) and a closing parenthesis. If you look closely, you will notice that each flow pattern has the same number of i's and o's as there are arguments in the predicate. This is no coincidence; each i or o corresponds to one of the arguments in the predicate.

An "i" flow pattern indicates that the argument specified has an *input flow*, while an "o" shows that the argument has an *output flow*. Each time you call a system predicate, you must follow one of the flow patterns given for that predicate.

The idea of a flow pattern is not hard to grasp. An input flow means that the argument in the position indicated must be bound to a value when the predicate is called. On the other hand, an output flow means that the predicate requires a free variable in that argument position (remember that a variable is free when it is not set to a value). To clarify this concept, think of an input argument as something you must supply to the predicate and an output argument as something you get back from the call. Understanding the way flow patterns work can be helpful when you are trying to figure out the different uses of built-in predicates.

A predicate that has a single flow pattern can only be used in one way. For example, although the built-in predicate `write()` can take a varied number of arguments, they must all be bound (have an input flow) before the call to `write()` can be made. Because of this, `write()` really has only one function: It outputs values.

Predicates with multiple flow patterns have more than a single function. Take, for example, the built-in predicate `makewindow()`, which has two flow patterns. In one version, all the arguments must be bound to a value, while a second variation allows `makewindow()` to be called with free arguments. This can be seen by looking at the flow patterns documented for `makewindow()` in the *Turbo Prolog Reference Guide*. When a call is made to `makewindow()` with an all output argument list, the variables will return the values of the currently active window.

Here is how this call can be made:

```
makewindow(Num, Att, Frame, Msg, Row, Col, R_Length, C_length)
```

Remember, all the variables must be free before the call can be made; otherwise, an error message will be given showing that you are using an illegal flow pattern for a standard predicate. As a test, load Turbo Prolog and clear the Editor window. Now, without a program in the Editor window, give the command to run a program. You will notice that the Goal: prompt quickly appears in the Dialog window. Type the call to `makewindow()` just shown and you will be surprised when the specifications for the Dialog window return as this call is made.

Calling built-in predicates from the Goal: prompt is an easy way to see how specific built-in predicates work. You can test any predefined predicate in this way; simply run an empty program and make a call to the built-in predicate of your choice. Try experimenting with different system predicates, using the various flow patterns shown in the Turbo Prolog guide.

Domains Section

The pieces of information that a computer program deals with are not all alike. Sometimes a program works with text, sometimes numbers, and sometimes with graphic figures or symbols. Within such categories there are more specific *types* of information. Text may come in the form of single characters or may consist of com-

plete phrases or sentences. Simple integer numbers (such as 1, 2, or 646) are quite different from sophisticated real numbers (such as $3.4 * 10e8$, where “e8” stands for “to the eighth power”).

Information Types

Different types of information are not just philosophically distinct. They are also handled differently inside the computer; individual types of information take up different amounts of computer memory. A value can be handled more efficiently by a program routine if the type of data is known in advance. For this reason, it is helpful to specify the domains of the objects that a predicate will use. For instance, if a predicate takes some “thing” as an argument, and it doesn’t know what type of information that thing can be, the program must make a broad definition and set aside the maximum amount of memory for the unknown object. Not only this, but the program must work with the most general-purpose routines to handle this thing.

If, instead, the program knows that the thing can only be an integer number, it can set aside the limited amount of memory demanded by an integer, then set itself up to efficiently work with that domain type. Such typing leads to smaller programs that run faster.

Type Checking

When a program is told that an object can only take a particular type of data, a message will alert you if a wrong data type attempts to occupy that space. Although the declaration of data types may seem like one extra step to do, the time spent in organizing your program will actually save time in the long run. This data analysis (or *type checking*) will help you to avoid situations that lead to trouble, such as when the wrong type of information is fed into a program.

A computer language can employ type checking to improve program speed, use computer memory more efficiently, and check for programming errors. When a program is compiled, it is checked to be sure that each argument in every predicate receives only the data type it was declared to take.

Declaring data types has its advantages. Because of the typing required, a programmer must carefully think out in advance what types of information will be passed and used in the program. The result is a more organized and clearly understood program, and one that requires less time spent with debugging.

Standard Domains

As with predicates, there are also two kinds of domains in Turbo Prolog: *standard domains* and *user-defined domains*. Standard domains are defined by the system, and include symbol, string, integer, real, and character. If only standard domains are used, the Domains section can be left out of a program.

User-defined domains come in handy when you want to be sure, by type checking, that only the required type of data is passed to a particular predicate. Creating your own domains also helps with documenting your predicate declarations and provides a way to declare data types that are not defined by the system. The standard domains are defined below.

Symbol The word "symbol" is used for this domain. An argument whose domain is a symbol may be represented in two different ways:

1. A group of consecutive letters, numbers, and underlines, beginning with a lowercase character. Here are some examples:

```
fred  
freds_car  
the_thing_in_the_cave  
respiratory_rate  
interest_rate_minus_inflation
```

2. A group of consecutive characters (letters, numbers, underscores, and spaces) that begins and ends with quotation marks (“”). This type is useful if you wish to begin the symbol with an uppercase character or include spaces within the symbol’s name. For example:

“freds car that I drove into a tree”

“University of Michigan Alumni”

Symbols and strings may look similar, and many argument values can be used with either domain, but Turbo Prolog interprets them quite differently. A symbol takes up less memory than a string, and symbols make matches more quickly when a program is run. The reason is this: Symbol *addresses* are kept in a reference table and are made to represent (or symbolize) objects in the program. This makes for very fast matching and compact memory storage. Strings, on the other hand, are examined character by character when a match is attempted.

String The word “string” represents the string data type. Unlike the second type of symbol introduced above, strings may contain any group of consecutive characters, beginning and ending with quotation marks (“”). Here are a few examples of valid Turbo Prolog strings:

“fred”

“Fred earns \$500 per week”

“When in the course of human events it becomes necessary to”

When you write out information (either in reports or to the computer screen), strings will often be your means of communication. To provide a flexible form of communication, strings may contain *control characters* that allow you to format the text according to your liking. For further information on formatting strings and text, refer to Chapter 10, which describes text output.

Symbols are prohibited from being more than 255 characters long, but the strings that Prolog reads in from a file can be up to 64K characters in length—a string variable can contain an entire file! For this reason, it is sometimes better to use strings instead of symbols as arguments to predicates.

Integer The word "integer" declares that an argument will take this data type. Any whole number from (and including) $-32,768$ to $32,767$ can be of the type integer. Arguments that are declared to be integers may contain these values. For example:

9
-200
21444

The limit placed on integers is imposed because of the personal computer's internal numbering system. To obtain the maximum efficiency when handling whole numbers, integers are stored as 16-bit values (a bit is the smallest piece of information a computer can handle, either a 1 or 0). In the personal computer's numbering scheme, 15 of the bits represent the actual number, while bit 16 represents the sign of the number (positive or negative).

Real The word "real" defines an argument as a real data type. A number with a real type can hold a value in the range of $1e-307$ to $1e308$, positive or negative. The format includes these options: sign, number, decimal point, fraction, e (indicating a power exponent of 10), a sign for the exponent, and the exponent itself. Unlike integers, a real value can contain a fraction of a whole number. Real numbers can range from very short to very long-winded. Here are some examples:

3
3.0
-3.1415926525e-218

Character The character domain type is represented by the word "char". Any single character from the standard ASCII table that is positioned between two single quotation marks (') can be a char. Characters in the upper range of the ASCII table (characters from 128 to 255) can be represented by a backslash (\) followed by the character's ASCII number value (all placed between single quotation marks). For example:

't'
'X'
'&'
'\129'

Creating Your Own Domains

The Domains section of a Turbo Prolog program is the place where you declare customized domain types for your program. There are several reasons why you may wish to do this. For one, not all Prolog data types are represented by standard domains. Also, using your own domains helps to document the declarations made in the Predicates section. Domain declarations can range from simple to complex, and they can be either local or global. This section introduces simple domain declarations.

Since predicate declarations must show the argument domains the predicate takes, using customized domains will help to clarify your program. For example, suppose you wish to write a set of facts describing the clients in your business. You might decide that a client fact will have four arguments: a client's first and last name, account number, and phone number. The predicate declaration you come up with might look like this:

```
Predicates  
  client(symbol, symbol, integer, string)
```

By making use of user-defined domains, this declaration could be improved to look like this:

```
Predicates  
  client(first_name, last_name, account_number, phone)
```

In effect, you have created documentation for your program, because it is now clear what values are used in the predicate `client()`.

To achieve this predicate declaration, your Domains section must look like this:

```
Domains
  first_name, last_name = symbol
  account_number        = integer
  phone                 = string
```

Notice that it is possible for several domains to be defined as the same type. This is done by separating the user-defined domains by commas, then setting them equal to a single domain type. In this example, all the domain types (the part on the right side of the equal sign) are standard domains. This may not always be the case, as will be apparent when complex declarations are introduced.

There is one catch with using customized domains in all of the predicates you declare. If you are not careful, the type-checking system will grind your program to a halt in order to issue a *type error*. Remember that when passing arguments from one predicate to another the arguments passed must be from the same domain. In the client example this can be helpful. A type error will be given if you attempt to place a first name where a last name should go. But in more complicated programs, using more general domain types will save you when you are furiously passing arguments from one predicate to another.

Goal Section

With most implementations of Prolog, the programs written must be run from within the Prolog environment. These implementations of Prolog are called *interpreted* because Prolog must interpret each line of program code as it comes to it.

Turbo Prolog is an excellent applications language because of its ability to create *stand-alone* programs from the code you write. Up until now, you have been running your programs from within the Turbo Prolog environment. But soon you will have the power to create programs that run directly from the DOS command line. All that needs to be done is to create a Goals section inside of your program, then compile your program to an .EXE file. (.EXE is the

extension given to a program that can be run from DOS. If this is new to you, refer to your DOS manual for more information about DOS file names and their extensions.)

The Goal section is analogous to a rule defined in the Clauses section of your program. The only differences between the two are that the goal is automatically called when your program starts and that no “:-” symbol is located after the keyword “Goal.” Take a look at the following program:

```
Domains
  person = string
  thing  = symbol

Predicates
  likes(person, thing)
  has(person, thing)
  wants(person, thing)

Clauses
  likes("Sally", shoes).
  likes("Eric", corn_dogs).
  likes("Leo", ice_cream).
  likes("Rebecca", roses).

  has("Eric", shoes).
  has("Sally", car).
  has("Dan", hammer).
  has("Ed", backpack).
  has("Rebecca", roses).
  has("Eric", corn_dogs).

  wants(Who, What) :-
    likes(Who, What) ,
    not( has(Who, What) ).
```

When this program is compiled and run, you can give the goal
wants("Sally", What).

to find out what Sally wants. The system will respond with a single solution, indicating that Sally wants a pair of shoes. However, with the use of a Goal section, this query can also be placed inside the program. Add the following code to the bottom of the program:

```
Goal
  makewindow(1,2,3,"",0,0,25,80) ,
  wants("Sally", What) ,
  write("Sally wants: ", What) ,
  nl.
```


Now when run, the program will not require you to input a query to start the processing. Notice, however, that to get a response out of the program, you need to actually write the solutions out. While queries given at the Goal: prompt will find and output all the solutions to the variables in the query, internal queries require that you output the values that you find.

If Sally wants more than just a pair of shoes, you will have to force Prolog to look for the other solutions. (The idea of forcing Prolog to backtrack to find alternate solutions is a concept that is covered in Chapter 7.)

The Goal section can have as many subgoals as you wish, each one separated by a comma and the last one ending with a period. When all the subgoals in the Goal section have succeeded, the program will terminate (it will also terminate if the Goal section fails).

It is common to see a Goal section with a single subgoal, as in

```
Goal
run.
```

In this instance, the program will start by making a call to the predicate `run()`, which is located in the Clauses section. When the predicate `run()` succeeds (or fails), the program will end.

You now have the power to create stand-alone executable programs. Simply include a Goal section in your program with the series of subgoals that you wish for the program to process. After this, compile your program to an .EXE, and you're on your way!

Advanced Program Sections

The four program sections that have been outlined form the backbone of any Turbo Prolog program. However, they are not the only sections you can use in your programs. Larger, more complex programs require that the Prolog source code be split up into separate *source code modules*. Doing so means that you will need to use the Global Predicates section to declare that certain predicates are glo-

bal to all modules. The arguments that are supplied to the global predicates must be defined in terms of global domains.

Many programs make use of Turbo Prolog's databases, both internal and external. Using this feature of Turbo Prolog requires that you make use of the Database section. Turbo Prolog version 2.0 also added the ability to declare constants in your programs. The constants that you declare must be done in the Constants section.

Global Declarations

User-defined predicates can be either local or global. A *local predicate* is a predicate that is defined in the same module in which it is declared. A *global predicate* declaration, on the other hand, declares that the predicate is defined in a separate source code module but can be accessed from any module.

Up until now all the predicates you have written are local predicates; no special section is needed to declare them. Global declarations are different; they must be declared in the Global Predicates section.

Modules and modular programming are advanced techniques that are used when a program becomes too large to compile as a single program. In such a case you must break up your program into separate source code modules and compile the modules separately. Once each module has been compiled, they can all be linked to form a single, stand-alone executable file. The use of modules requires that a project be defined and your program compiled in a special way. These programming techniques are discussed in more detail in Chapter 13.

Constants Section

The ability to declare and use symbolic constants in a program is a feature new to Turbo Prolog version 2.0. Constant declarations are global to the entire program, meaning that once declared the value

of the symbol defined is constant and is in effect for all parts of the program.

The Constants section is used to set symbols to specific values. This makes it easy for a program to be modified after it has been written. Take, as an example, an interest rate program. Without constants, every time the interest rate fluctuated, you would have to go into the program and change every occurrence where the interest rate is used. With constants, you can set up a symbol to hold a specific value. Changing the interest rate in this case would be as easy as changing a single value. A program may have as many constants declared in it as is necessary.

Constant declarations look like this:

Constants

```
ideal_class_size = 15
strange_number = 47.2 / (24 * (18 - 11))
student_origins = [france, brazil, japan]
pi = 3.14159
```

Each constant declaration is a single line long, and each new declaration must begin on a new line. A new line tells the compiler that one constant declaration is finished and the next one is on the way.

After making the constant declarations, you can employ the names of the constants in the body of your code. When the program is compiled, the constant names will automatically be replaced with the values that the names represent.

In order for the compiler to replace the constant symbols with their respective values, constants must begin with a lowercase letter. This is to keep the compiler from mixing up constants with variables. You may, however, capitalize the constant names in the declarations if you wish. The declaration `Ideal_size` actually sets up the symbol `ideal_size` to be recognized as the constant. Capitalizing constant declarations is not advised; why confuse the readability of the program in this way?

The only other regulation about constant declarations is that they cannot call themselves. *Recursive definitions* are not allowed because there is no way for the compiler to resolve the value of such a constant. For instance, the recursive definition

Constants

```
required_return = 0.09*(payment/interest)+required_return
```

would send the computer into an infinite loop while it tried to figure out the actual value of `required_return`.

Database Section

The Database section of your program is reserved for the declarations of your “dynamic database” facts. The ability of a Prolog program to add to and retract from its knowledge base while running gives Prolog an edge over other programming languages. If, while a program is being processed, a new item can be proven true, then this fact can be added to the group of facts that the program knows to be true. Likewise, if an existing fact proves no longer to be trustworthy, it can be removed from the program’s knowledge base.

Like local and global predicates, database facts can also be declared to be local or global. The same logic presides for global database facts as for global predicates. A global database fact can be accessed from any program module, while a local database fact is local to the module it is declared in.

Advanced Declarations

Declaring simple predicates and domains is a fairly straightforward process. However, these simple declarations are not the only ones that are allowed in Turbo Prolog. Turbo Prolog also allows for the declaration and use of compound data types and complex data structures. While complex structures are left to a later section in the book (Chapter 9 deals with lists and recursive structures), compound objects can be put to good use in your programs right away.

Compound Objects

Sometimes it is nice to be able to handle a set of values as a single object. For example, if you are keeping a database of purchases, the

date that each purchase was made must be kept as three separate arguments. With the use of compound data types, these three values can be grouped into a single argument. *Compound objects* look like relationships within relationships. Here's an example:

```
purchase (stove, date(12,16,1985), 459.85)
```

Actually, a compound object is just another Prolog term. The object `date()` is called a *functor* and is handled as a single argument. To declare a compound object, you declare the functor and the arguments that go along with the type in the Domains section of your program. Here is an example of how the compound object `date` can be declared and used in a predicate declaration:

```
Domains
  d = date(integer, integer, integer)
  price = real

Predicates
  purchase( symbol, d, price)
```

The arguments inside the functor can also be user-defined domains, as shown here:

```
Domains
  year, month, day = integer
  d = date(day, month, year)
```

The power of using compound objects will become evident as you begin to use them. For example, you can pass an entire data object as a single variable. With the unification built into Turbo Prolog, the argument `date(12,16,1985)` can make the following matches:

```
X = date(12,16,1985)
date(Month, Day, Year) = date(12,16,1985)
date(—, —, Year) = date(12,16,1985)
```

This may seem cryptic now, but here is a program that shows how compound objects can be put to work:


```

Domains
    name      = n(first_name, last_name)
    record    = p(amount, service)
    amount    = real
    first_name, last_name, service = string

Predicates
    billing_record ( name, record )
    process_record ( name, record )
    print_report

Clauses
    billing_record(n("Freddy", "Benagan"),
                  p(54.95, "printed checks")).
    billing_record(n("Sammy", "Fleetman"),
                  p(109.22, "developed film")).

    print_report :-
        billing_record(Name, Record) ,
        process_record(Name, Record).

    process_record(Name, p(Amount, Trans)) :-
        Name = n(First, Last) ,
        write("We owe ", First, " ", Last) ,
        write(" $", Amount, " for: ", Trans).

Goal
    makewindow(1,2,3, " Accounts Payable ", 0,0,25,80) ,
    print_report, nl, nl ,
    write("Press any key...") ,
    readchar(_).

```

Multiple Domain Types

It is possible to declare a single user-defined domain that consists of more than one domain type. Although you cannot declare a domain equal to more than one standard type, it can be useful to create a domain that can be one of several objects. This is how it is done:

```

Domains
    item = book(string); magazine(string)
Predicates
    bought(item)

```

Here the predicate `bought()` can take either of the compound objects declared. To create a domain that can relate to one of several

objects, separate the different objects with a semicolon on the right side of the domain declaration. Remember that the semicolon in Prolog represents OR, so the declaration above can be read as "The domain item can take on the value of book(string) OR magazine(string)."

Lists and Recursive Data Types

A *recursive object* is an item that is defined in terms of itself. A *recursive data object* is a structure that has part of itself defined as an object like itself. Prolog is the only common programming language that provides for recursive data types. Although the concept is rather advanced, here is an example declaration of a recursive data object:

```
Domains
  tree = t(tree, index, tree); empty
  index = integer
```

In this declaration, the domain "tree" can either equal a compound object or it can equal the symbol "empty." An object that is declared to be of this type could take on the following complex value:

```
t(t(empty,1,t(empty,2,empty)),3,t(t(empty,4,empty),5,empty))
```

This data structure is known as a tree, where the index nodes on the left side of the tree must be less than the index nodes on the right side of the tree. This data structure is the basis of binary trees, a structure used to store and retrieve data.

When setting up predicates, you will sometimes come across a situation in which you cannot tell ahead of time how many values a particular argument will take. For example, if you are creating a program that keeps track of golf scores, certain players will have several to record, while others will have only a few scores or none at all. A separate fact can be written for each individual score, but a more elegant way of representing the data would be to use list notation.

Lists are recursive data structures that may have zero or more elements. A list does not have a predefined system domain, so the user must create a domain type to represent a list. For example, the following declares a list structure:

```
Domains
    integerlist = integer*
```

Complex data types, along with a look at their uses, will be covered in Chapter 9.

List of Turbo Prolog Program Sections

The list that follows contains all the Turbo Prolog program sections and the order in which they must appear in a program.

- Compiler Directives
- Global Sections
 - Global Domains
 - Global Database
 - Global Predicates
- Constants Section
- Domains Section
- Predicates Section
- Clauses Section
- Goal Section

All global declarations must come before the local declarations. A program may contain more than one Domains, Database, and Predicates section, although a predicate or domain must be declared before it is used in the program code.

7 *Controlling Prolog's Search*

After having played a little with Prolog, you will see that the most efficient path to solving a problem is not always the one followed. Sometimes a program will output the same solution over and over, while at other times the path chosen will be the wrong one entirely. Your program might be caught in an infinite loop or you might receive the error message "heap overflow," indicating that your system has run out of available memory. No matter how frustrating these situations may seem, you cannot blame these flow control problems on Prolog. Prolog just doesn't know any better. It is up to you, the programmer, to provide the proper flow control so that the program can find the right solution and execute it in a reasonable amount of time.

Although some "logic programmers" feel that controlling the search of a program takes away from its declarative aspect, it is often necessary nonetheless to prod the direction of flow when developing an application that must follow a sequence. Even though some programs require that you not bother Prolog in its search, procedural programs usually don't need the detailed level of reporting that Prolog tends to offer. The ability to control the search in a program allows Prolog to be used as a general-purpose language as well as a language for solving difficult logic problems.

Control Structures

The idea of flow control is nothing new. If you have programmed with another language you should be well acquainted with looping and control structures. If Prolog is your first programming language, then control structures can be easily linked to the way you make decisions in everyday life. A *control structure* can be thought of as the way to perform a specific task or go about making a particular decision.

Looping

Let's say that you have a stack of letters to respond to. To get the letters answered, you sit down and perform a "loop" until all the letters are written and are off your desk. With one type of loop (or control structure), you write one letter at a time until no more letters are left to respond to. This looping structure can be thought of as a *conditional loop*. You repeat the process until the condition is met that there are no more letters left to respond to. As you finish each letter, you check to see if any more letters are left in your in-basket. If there are, then the whole letter-writing process starts over again from the top. At last, when you go to check for a letter and none are left, you are done with the task. The outline of such a loop would look like this:

```
If MoreLetters = yes
    get_a_letter (Letter)
    write_a_letter (Letter)
loop
```

Another type of loop can be used if you know in advance how many times the loop must execute. If, by chance, you know that there are 23 letters on your desk, a specific stopping condition can terminate the looping process. You can stop responding after the twenty-third letter has been taken care of. This type of loop can be thought of as a *counter loop*. A counter is used to keep track of

each loop performed. Before each letter is started, a comparison is made between the counter and the number of letters that need to be written. The outline of this structure is

```
for 1 to 23
  get_a_letter (Letter)
  write_a_letter (Letter)
loop
```

Variations of these two looping structures exist, but the structures shown here are models of the basic control forms used to complete a repetitive task.

When You Have a Choice

Another form of program control is the **IF-THEN** structure, a logical construction that barely needs an introduction. If a condition (or set of conditions) is met, then an action can take place. Using an everyday example, suppose that you are going to buy some fruit. If the fruit is fresh, then you will purchase it.

Adding to the **IF-THEN** structure, **IF-THEN-ELSE** provides an alternate action when the **IF** condition is not met. This structure allows for one of two separate actions to be taken, depending on the outcome of a test. If the fruit is fresh, then you will buy it. Otherwise, you will buy crackers and cheese.

If it is possible to take more than two actions that depend on a condition, then a **CASE** structure (as it is called in Pascal) is needed. For example, suppose you are at a restaurant and it is time to choose an item from the menu. There are several items to choose from, but only one can be selected. Your choice will most likely be based on two conditions: what you feel like eating and how much money you have with you. You must choose a single action out of several, based on a set of conditions.

Actually, the structures **IF-THEN**, **IF-THEN-ELSE**, and **CASE** all perform the same operation: They each execute an action (or series of actions) based on the outcome of a test. The only difference between these structures is the number of possible options available

for a given circumstance. With **IF-THEN**, there is only one possible action; if a test succeeds the action is taken. **IF-THEN-ELSE** allows for a second action to be specified if the test does not succeed, and **CASE** allows for as many actions to choose from as are needed.

Most procedural programming languages provide control structures by building them into the language. In Pascal, a **REPEAT-UNTIL** structure, a **DO-WHILE** structure, and even a **FOR-NEXT** structure are built right into the language. In C, the **CASE** structure is known as a *switch*. The Prolog language does not provide any built-in structures for controlling the processing of a program. In trade for this, Prolog provides both an inference engine and a matching mechanism that makes it easy for you to create the structures needed to control a program.

Prolog's Control Predicates

Turbo Prolog contains two built-in predicates that allow you to control the processing of a program. They are *cut()* and *fail()*. The *fail()* predicate forces Prolog to backtrack, looking for alternate solutions to a query. A completely opposite predicate is *cut()*, which is used to prevent backtracking. Indicated as “!”, a cut narrows down the search space of a given problem. In addition, a cut can tell Prolog that it is on the right path, so that it will not look back for other solutions to the problem at hand.

A third predicate that is sometimes used for flow control is the predicate *true()*. The function of this predicate is fairly straightforward: It instantly succeeds and it leaves no backtracking points. You may wonder what possible use such a predicate may have, but in logic programming it can turn out to be quite useful.

Using Fail to Repeat a Process

Chapter 5 showed that backtracking is an essential element in Prolog's search strategy. When a call is made to satisfy a subgoal, the search goes through the clauses looking for a solution to the query.

When a dead end is arrived at, the program backs up just enough to find another path that can be searched for a possible solution. This backing up is known as Prolog's *backtracking mechanism*. The important thing to keep in mind is that backtracking kicks in every time Prolog comes to a road block. It is as if the program comes to a dead end. In reality, the block is simply the failure of a call to return true. Every time something fails, backtracking begins.

As an example, suppose you wish to look up all the clients in a database and process their records by age. The following program illustrates how this works:

```
Domains
  age, id_number, code = integer
  first, last          = symbol
  address              = string

Predicates
  client(id_number, last, first, address, age)
  go

Clauses
  go :-
    client(Id,_,_,_,Age) ,
    write("Client: ", Id, " Age: ", Age), nl ,
    fail.
  go.

  client(1341, slipman, fred, "101 Washington", 18).
  client(1282, handler, sylvia, "212 Alabama Way", 40).
  client(1099, rouzo, alice, "130 B Harvard Place", 37).
  client(1253, davis, tom, "110 Mott Street", 26).
```

Since there is no need to process the clients' names and addresses in this program, the anonymous variable is used in the call to `client()`. This is an ideal application for the anonymous variable; there is no need to create an extra variable name when the value it becomes bound to will never be used.

When run, the program will output each client's ID number and age, both found by processing the program. To see the program run, type the program name, and press F9 to compile it to memory. Type R to run the program, and give the goal `go` to begin processing.

To control the flow of processing in this program, a conditional loop is modeled. Using `fail()` in the `go()` predicate forces Prolog to backtrack through all the solutions to `client()`. One at a time, each client is looked up and its data processed. The loop terminates when no more clients are left to process. When this condi-

tion is met, the loop is finished and control returns to where `go()` was first called from.

Taking a close look at the `go()` predicate, you may notice the lack of a test to stop the looping process. Actually, an outline of this Prolog loop is

```
get_client_info (ID, Age)
write_info (ID, Age)
backtrack
```

The call to `fail()` induces looping by forcing Prolog to go back and look for other solutions to the problem. But remember how backtracking works. Prolog returns control to the last backtracking point placed, and attempts to redo the call made at that point. It is important to see that using `fail()` does not cause the loop to start over from the top, but instead forces Prolog to go backward, looking for alternate solutions.

Prolog will loop between `fail()` and `client()`, each time finding a new solution for the call to `client()`. Once Prolog finds a call that can be successfully redone, processing of the program can continue. In this case, the ID and age are printed out for all clients in the database.

After the last client has been processed, the first clause to `go()` fails because there are no more solutions to any of the subgoals in that clause. At this point, backtracking takes us back to the second clause defining `go()`. This clause will succeed, and in doing so, the original call to `go()` will succeed and returns true. As a test, remove the second clause defining `go()` and see how the program ends when run. This second clause can be thought of as a "dummy" clause. It will always succeed when called, and because of this the predicate `go()` can never fail.

Now replace the last clause for `go()` and add one more clause to the program to create a cleaner interface. Insert the following clause to the top of your program, making it the first clause in the Clauses section.

```
go :-
    makewindow(1,2,3," Client ID's ",0,0,25,80) ,
    fail.
```

The predicate `go()` now has three clauses defining it. In Prolog, a section of code in which several clauses create the definition for a single predicate is known as a *procedure*.

A Case Structure in Prolog

In Prolog, it is possible to write several clauses to define a single predicate. Since each clause can be made to execute when a different condition is met, it is quite natural to write a **CASE** structure in Prolog. Take the following Prolog procedure:

```
get_age_code(Age, Code) :-  
    Age < 21 ,  
    Code = 1.  
get_age_code(Age, Code) :-  
    Age >= 21 ,  
    Age < 40 ,  
    Code = 2.  
get_age_code(Age, Code) :-  
    Age >= 40 ,  
    Code = 3.
```

The predicate `get_age_code()` models a simple **CASE** structure in Prolog. This predicate is called with the first argument bound to an age and the second argument free. For example:

```
get_age_code(26, Code)
```

When called in this manner, the second argument will return a specific age code. As with a case statement, a single choice is made from several selections, depending on the value of the age.

Relating this to the control structures just discussed, the procedure could have one, two, or several clauses defining the predicate. For example, the following code represents an **IF-THEN** structure:

```
get_age_code(Age, Code) :-  
    Age < 21 ,  
    Code = 1.
```

If the age is less than 21, then the age code is 1. Within an **IF-**

THEN structure, nothing is provided for when the condition does not succeed. With an **IF-THEN-ELSE** structure, an alternate is given:

```
get_age_code(Age, Code) :-
    Age < 21 ,
    Code = 1.
get_age_code(Age, Code) :-
    Age >= 21 ,
    Code = 2.
```

The age code is 1 if the age is less than 21; otherwise, the age code is 2. These clauses are said to be “mutually exclusive” because only one clause can possibly succeed for any given age.

A full **CASE** structure is shown in the first program code where a single age code is chosen from several possibilities. With Prolog, there is no theoretical limit to the number of choices that can be made available. However, Turbo Prolog allows for a maximum of 500 clauses per predicate definition, which does limit the number of possible actions that can be provided.

You may have noticed that one control structure in Prolog remains to be discussed. Counters, and how to create loops using them, are given full attention in Chapter 13 when recursive loops are introduced.

Cutting Down on Search Time

Sometimes Prolog may spend valuable time searching for information that isn't needed to solve a subgoal. With Prolog, computers become quite vulnerable to the “combinatorial explosion” that can result from just a few levels of subgoals. Unwary programmers may be left scratching their heads as a program goes off to never-never land when the run command is given.

Atomic physics provides a good analogy for what can happen. An atomic bomb is based on the principle of a fission chain reaction: A flying neutron rams and breaks up a radioactive atom, which results in a lot of energy and two flying neutrons. These two neutrons then ram into other atoms, generating four neutrons. Four neutrons now smash into four more atoms, generating eight flying neutrons. Such a chain escalates logarithmically, and soon millions of neutrons are smashing into atoms.

A computer program can behave in much the same way. If a call leads to a subgoal, which in turn leads to two subgoals, and each subgoal then leads to two more subgoals, the number of calls needed to satisfy a query can quickly multiply to impractical levels. Such a chain reaction can quickly overwhelm a computer and lead to huge increases in processing time. In this situation, you need to help Prolog cut down on the search space of the program.

To begin with, you can write the body of your rules in a way that makes for the most efficient search. Only practice and an understanding of what is needed to solve a problem will accomplish this task. A general rule of thumb is to arrange the subgoals in your rules so that the clause will fail as quickly as possible. If a clause is going to fail, you want to get it over with so that you can go on to a clause that may provide an actual solution. This can be accomplished by placing the most general test at the top of the list of subgoals.

A simple example illustrates this point. Suppose you are writing a clause to define a father relationship. You can go about the search by first finding a parent, then testing to see if this parent is a male. The Prolog rule looks like this:

```
father(Father) :-  
    parent(Father, Child) ,  
    male(Father).
```

But this code does not prove to search effectively. Since half of the parents will end up being females, the code as written will spend valuable time looking up parents who are mothers. The search for a father can be optimized by placing the test for a male before trying to look for a parent.

Another way to cut down on search time is to let Prolog's matching mechanism do some work. Take the **CASE** structure that determines an age code:

```
get_age_code(Age, Code) :-  
    Age < 21 ,  
    Code = 1.  
get_age_code(Age, Code) :-  
    Age >= 21 ,  
    Age < 40 ,  
    Code = 2.  
get_age_code(Age, Code) :-  
    Age >= 40 ,  
    Code = 3.
```

The predicate `get_age_code()` is called with the first argument

bound and the second argument free. However, by using the power behind the built-in matching mechanism, the code can be simplified to

```
get_age_code(Age, 1) :-
    Age < 21.
get_age_code(Age, 2) :-
    Age >= 21 ,
    Age < 40.
get_age_code(Age, 3) :-
    Age >= 40.
```

These two sections of code are identical in function but different in structure. In the second procedure the age code is set to a value in the head of the rule. This works because the age code is returned only if the body of the rule succeeds. When one of the rules succeeds, the age code in the head of the rule can be returned to the clause that called `get_age_code()` to begin with.

Using the matching mechanism to perform tests is also a good way to optimize program code. If, for instance, the age of 21 has a specific age code, you can write it into Prolog like this:

```
get_age_code(Age, 1) :-
    Age < 21.
get_age_code(21, 5).
get_age_code(Age, 2) :-
    Age > 21 ,
    Age < 40.
get_age_code(Age, 3) :-
    Age >= 40.
```

An age code of 5 is assigned to anybody who is exactly 21 years old.

Cut (!)

Arranging code so that it will test effectively is only one way to cut down on search time. Another approach lets Prolog know that a section of the search space can be skipped altogether. This not only saves computing time, but also makes it easier to follow the flow of the program when backtracking kicks in.

The cut is essentially a built-in predicate with no arguments. When called, the cut instantly succeeds and processing goes on to the next call in line. The cut is very useful because, once it has been

passed, backtracking cannot return to a point prior to the cut in the clause being processed.

The power of this process can be shown by using the familiar `get_age_code()` procedure. Once you have determined a valid age code, there is no point in searching the other `get_age_code()` clauses. Since they have been written to be mutually exclusive of each other, only one clause can possibly succeed for any given age.

Once the correct age code clause has been determined, a cut can prevent backtracking to alternate `get_age_code()` clauses. For example, if age 15 is being processed, an age code of 1 will be returned. But after the return, a backtracking point will be placed next to `get_age_code()` because there are still two clauses that may provide alternate solutions to the call. Although neither of the other clauses can return an age code, Prolog has no way of looking ahead, and therefore it cannot determine ahead of time that further searching will end up as futile time spent. Here the cut can help out ("cut" is represented by "!"):

```
get_age_code(Age, 1) :-
    Age < 21, !.
get_age_code(21, 5) :- !.
get_age_code(Age, 2) :-
    Age > 21,
    Age < 40, !.
get_age_code(_, 3).
```

Once the proper age code is found, a cut is passed. Because of this, backtracking points will not be placed to mark the untried `get_age_code()` clauses.

Notice that the fourth clause acts as a catchall. There is no need to perform a test because any age equal to or greater than 40 will return an age code equal to 3. With such a catchall, it is easy to provide an error-trapping routine in **CASE** statements. For example:

Predicates

menu

process_choice(integer)

Clauses

menu :-

```
    makewindow(1,2,3," Menu ", 10,10,10,60) ,
    nl, nl ,
    write(" READ  (1)"), nl ,
    write(" WRITE (2)"), nl ,
    write(" SORT  (3)"), nl, nl ,
    write("Choose an action: ") ,
    readint(Choice) ,
    clearwindow ,
    process_choice(Choice).
```



```

process_choice(1) :-
    !, write("You chose the read option.").
process_choice(2) :-
    !, write("You chose the write option.").
process_choice(3) :-
    !, write("You chose the sort option.").
process_choice(_) :-
    write("NOT A VALID CHOICE!").

```

This small program shows an archaic way of creating a menu. The user is prompted to enter a number according to the desired action. If the number entered corresponds to one of the menu items, then the appropriate action is taken. If the user enters an incorrect choice, then an error message is displayed.

The cut plays an important role in this structure because without it the error clause always has an option to be executed. The catchall clause in a **CASE** statement never contains a test because it is assumed that an error has occurred if none of the other clauses have succeeded. Without these cuts, the error-trapping clause can be returned to if something fails later on in the program, even though no error has occurred.

A Detailed Look at Cut

Notice that the cut affects only the predicate that contains it. Once passed, the cut prevents backtracking to subgoals placed before the cut, and it also prevents backtracking to alternate clauses that define the same predicate being processed. The code that follows is more conceptual than it is functional, but should help to clarify how the cut works:

Predicates

```

a
b
c ( integer )
d ( integer )
e ( integer )
f ( integer )

```

Clauses

```

a :- b, write(" Success! ").
a :- write(" Failure. ").

b :- c(X), d(X), !, e(Y), f(X).
b.

c(1).  c(2).  /* c(3). */

```



```
d(3).  
e(1).  e(2).  
f(1).
```

Although this example may seem complicated, it is made to detail all the effects of placing a cut. Once you fully understand this example, you will have mastered the most confusing aspect of Prolog programming.

Here's how this program works. The processing begins with a call to `a()`. This call outputs "Success!" if `b()` can be proven true or "Failure." if `b()` fails. When the first clause defining `b()` fires, `c()` is called and returns with `X` bound to 1. Next, a call is made to `d(1)`, which quickly fails. Since the cut has not been passed, backtracking can bring Prolog back to redo the call to `c()`. This time, `X` returns bound to 2, and `d(X)` is again tried, with `X` equal to the value 2. Again a failure ensues, and Prolog must backtrack to the second clause defining `b()`. Here, the second clause to `b()` succeeds and a return is made to `a()`. As a result of this, "Success!" is written out.

Now, try the program again, but this time remove the comment markers surrounding the third clause defining `c()`. When this is done, the call to `d()` will succeed when `X` becomes bound to 3, and the cut will then be passed. After this, a call is made to `e()`, which returns with `Y` bound to 1. Processing now moves on to the call to `f(3)`, which ends up in failure because there is no fact that can prove it true. Backtracking can take us back to the call to `e(Y)`, but even when `Y` is bound to 2 the call to `f(3)` still fails. Notice that backtracking can still take place within a predicate after a cut has been passed. It is not possible, however, to return to subgoals placed before the cut.

Because the cut has been passed, Prolog cannot look for an alternate solution to either `c()` or `d()`. Also Prolog can't try an alternate clause defining `b()` because the cut was passed in the `b()` predicate. So the call to `b()` fails, and Prolog must backtrack to the second `a()` predicate, printing out "Failure." as it does so.

The cut is the most easily abused feature in Prolog. However, once the consequences of placing a cut are completely understood, you will find yourself removing the word `trace` from the top of many programs. Return to this example often. After you can follow the logic of these clauses without hesitation, you will be able to place and use the cut with complete confidence.

Determinism and Nondeterminism

In Prolog, when a call is capable of generating multiple solutions through backtracking, that predicate is said to be *nondeterministic*. An example of this is the user-defined predicate `client()` shown in the first program in this chapter. In contrast to this is a predicate that can generate only a single solution when called. A predicate with this property is known as *deterministic*. The predicate `get__age__code()` is an example of a deterministic procedure.

When defining predicates it is important to know whether the predicates you write are deterministic or nondeterministic. If you know the status of your predicates it will be easier to follow the flow of processing through your program. For example, if you know that all the clauses in a program are deterministic (they can only return one solution to a given call), you will also know that the program will terminate if any one condition fails. Since none of the predicates has the ability to generate alternate solutions, the program will fail if any one of the subgoals cannot be proven true. If, on the other hand, you know that a predicate is nondeterministic, then you will also know where Prolog will return to when backtracking begins.

When creating loops that are triggered by a call to `fail()`, it is extremely important to know which predicates are nondeterministic within the loop. If there are any nondeterministic predicates between `fail()` and the top of your loop, you may be surprised when your loop comes up with unpredictable results. Keep in mind that when a call fails, Prolog returns to the last nondeterministic call made and attempts to redo that call. Be sure that the call at the top of your loop is nondeterministic, so that Prolog can come up with an alternate solution, then go back down into the body of the loop.

Use the `cut` to create predicates that are deterministic. Remember that once a `cut` is passed in a predicate, Prolog is not able to backtrack to any other clauses defining that predicate. In essence, a clause that contains a `cut` defines that predicate as deterministic (although the `cut` must be passed in order for it to have any effect). In any code you write be sure that deterministic predicates do not generate backtracking points. This way your program will operate with maximum efficiency.

For example, adding cuts to the predicate `get__age__code()` creates a predicate that does not leave any backtracking points.

Without the cuts, the predicate is still capable of returning only one solution (it is deterministic), but that does not prevent Prolog from placing backtracking points when this predicate is called.

The following program uses the code shown at the end of Chapter 6 to incorporate the ideas covered so far. As a test, follow this program through in Trace mode, and decide where a cut (or cuts) can be placed to optimize Prolog's search.

Domains

```
name      = n(first_name, last_name)
record    = p(amount, service) ; r(amount, service)
amount    = real
first_name, last_name, service = string
```

Predicates

```
billing_record ( name, record )
process_record ( name, record )
print_report
```

Clauses

```
billing_record(n("Freddy", "Benagan"),
               p(54.51, "printed checks")).
billing_record(n("Sammy", "Fleetman"),
               p(109.22, "developed film")).
billing_record(n("Amanda", "Sterling"),
               r(554.95, "Landers film")).
billing_record(n("Glen", "Woodson"),
               p(109.22, "film")).
billing_record(n("Susan", "Stanson"),
               r(554.95, "Stanson film")).
billing_record(n("Ramies", "Corp."),
               p(366.72, "film titles")).
billing_record(n("Ralph", "Mayers"),
               r(554.95, "Mayers film")).
```

```
print_report :-
```

```
    billing_record(Name, Record) ,
    process_record(Name, Record) ,
    fail.
```

```
print_report.
```

```
process_record(n(First, Last), p(Amount, Trans)) :-
    shiftwindow(1) ,
    write("We owe ", First, " ", Last) ,
    write(" $", Amount, " for:\n   ", Trans, ".\n").
```

```
process_record(n(First, Last), r(Amount, Trans)) :-
    shiftwindow(2) ,
    write(First, " ", Last, " owes us:\n ") ,
    write(" $", Amount, " for: ", Trans, ".\n").
```

Goal

```
makewindow(1,2,3, " Accounts Payable ", 0,0,25,40) ,
makewindow(2,6,1, " Accounts Receivable ",0,40,25,40) ,
print_report ,
shiftwindow(1), nl ,
write("Press any key...") ,
readchar(_).
```


Prolog Clauses Versus Procedural Statements

When you write Prolog rules to create the control structures outlined in this chapter, keep in mind that the format of a Prolog clause is vastly different from a statement written in a procedural language such as C or Pascal.

In most procedural languages the “body” of a statement is executed if the condition in the “head” is met (there is actually no head or body in a procedural statement, but this analogy may help you to visualize a statement). In Prolog, the opposite is true: The head of a rule is executed (or proved true) if the conditions in the body are met.

Seen in this light, a Prolog rule is actually in the form of a **THEN-IF** structure, rather than the **IF-THEN** statement that so many programmers are familiar with. Failing to see this difference can cause many frustrating hours in debugging or in time wasted attempting to code a particular control structure.

A Backward-Chaining System

Prolog uses a backward-chaining search strategy when finding solutions to problems. When Prolog sets out to solve a problem, it may come across a rule that it needs to prove true in order to satisfy a query. When this rule fires, the subgoals in the body state the conditions that must be met to prove the call true. This can be seen as going backward to solve a problem; a statement is proposed and Prolog then tries to prove it true.

A forward-chaining system is exactly opposite: A series of conditions must be met before any action can take place. Once a checklist is taken care of, processing may proceed by executing the statements that follow. In this searching system, questions are first asked before any conclusion is presented. This is opposed to presenting a conclusion and attempting to prove it true.

It is helpful to understand the difference between these two search strategies. In Prolog, it is possible to model a forward-chaining search, but by default, Prolog will use the backward-chaining system in its search for solutions.

8 *Prolog Math*

Originally Prolog was conceived to solve problems that dealt with objects and the relationships between objects. The nature of these logistical problems meant that Prolog had to be especially good at manipulating the names or symbols that represented the objects in the problem at hand. In the beginning Prolog was designed with the idea that programs dealing with complex numerical equations would be left to other, more conventional programming languages. But Prolog's numerical clumsiness didn't last long. Programmers quickly realized the power of Prolog, and arithmetic facilities were soon added to the language.

With Turbo Prolog it is not necessary to use other languages to solve mathematical equations because it contains a full complement of trigonometric and logarithmical functions as well as various mathematical operators. These powerful features allow you to add mathematical equations with both integer and real numbers to your programs.

In Turbo Prolog, real numbers range from $1\text{E}-307$ to $1\text{E}+308$, and can have a sign, a mantissa, a decimal point, a fractional part, the E character (to set off the exponent), an exponent sign, and an exponent. However, only the whole number is mandatory. The sign, the decimal point, the fractional part, and all exponent parts are optional.

Integers in Turbo Prolog are whole numbers and range from $-32,768$ to $+32,767$. Integers are automatically converted to real numbers, and real numbers to integers, if the arithmetic operation requires it.

Numbers as Symbols

In Prolog numbers are handled no differently than the other symbolic values in the program. Mathematical predicates process numeric values instead of character values, and like other predicates, have the possibility for either succeeding or failing when called. The arguments to calls that are mathematically evaluated will either return a solution to the query or fail when a comparison is made.

Operator Position

In Prolog numeric values are objects to mathematical relationships. Because of this an addition can be written in the *prefix notation* that is common in Prolog. When prefix notation is used, the operator (or relationship) is placed in front of the arguments in the equation. Using this notation, a simple addition looks like this:

$+(3, 4, Z)$

Here the integer values 3 and 4 are added together and placed in (or compared to) the third argument in the predicate. This notation may seem unwieldy since prefix notation is not normally used in mathematical operations. However, a common prefix operator is the minus sign when it is used to signify a negative number.

An operation can also be written in *postfix notation* by placing the operator after the objects in the relationship:

$(Z, 3, 4)+$

An example of a postfix operator is the mathematical symbol that shows an exponent. Following a numeric value an exponent is shown when the character E is followed by a power of ten, as in 2.386E14.

To make you feel at home, Turbo Prolog uses *infix notation* to represent most mathematical operations. By using this type of notation (instead of the more cryptic prefix or postfix notations),

mathematical operations remain familiar. With infix notation a simple addition is written like this:

$$Z = 3 + 4$$

Arithmetic Operators

Table 8-1 lists the basic arithmetic operators used in Turbo Prolog. All operators use infix notation and must be placed between two numeric values of either real or integer type. The table also shows the data type resulting from the mathematical evaluation. If either operand in addition, subtraction, or multiplication is real, the result will also be real. In division, the result will always be real regardless of the operand type. The two operators “mod” and “div” always work with integers by the very nature of their function, and the unary operators (positive and negative signs) don’t affect the result data type at all.

Order of Evaluation

Prolog follows the standard mathematical practice of evaluating expressions from left to right, with multiplication and division having a higher priority than addition and subtraction. Parentheses may be used to override the default order of evaluation; anything inside of a set of parentheses will be evaluated first. All operations inside parentheses follow the usual left to right order of evaluation. Unary operators take an even higher priority than division and multiplication because the positive or negative status of a number must be known before a division can take place.

Mod and Div

The operators mod (short for modulo) and div both involve division. While it is only possible to use these operators with integer

Symbol	Operation	Priority ¹	Operand1 Domain	Operand2 Domain	Result Domain
+	Addition	1	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
-	Subtraction	1	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
*	Multiplication	2	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
/	Division	2	integer	integer	real
			integer	real	real
			real	integer	real
			real	real	real
mod	modulo division (result is the remainder of the division of operands 1 and 2)	2	integer	integer	integer
div	Div division (result is the quotient of the division of operands 1 and 2)	2	integer	integer	integer
+	Positive (unary) ²	3	integer		integer
			real		real
-	Negative (unary)	3	integer		integer
			real		real

¹“Priority” refers to the order in which operations are executed. Higher-priority operations are performed before lower-priority operations; the higher the number, the higher the priority. Operations within parentheses are performed before those outside parentheses, and operations are performed from left to right within an expression. Both the “mod” and “div” operators have priority 2, equal to a division.

²“Unary” refers to the use of the symbol as a specification of the sign of a single value.

Table 8-1. Standard Arithmetic Operators (Using Infix Notation)

numbers, their use is diverse. With infix notation both operators return a result based on the outcome of a division. While the oper- and mod returns the remainder from a division operation, div

returns the whole number gained from the result of a division. For example, these two statements are true:

$7 = 15 \text{ div } 2$

$1 = 15 \text{ mod } 2$

A common use of the mod operator is to find out which years are leap years. A leap year is any year that is divisible by 4, excluding centennials (except every 400 years). Using mod, it is possible to write a procedure to determine whether a year is a leap year. In the following program, the predicate `leap_year()` succeeds if the year supplied is a leap year; and fails if it is not a leap year:

```
Predicates
    leap_year(integer)
    check_centennial(integer)
Clauses
    leap_year(Year) :-
        0 = Year mod 4,                % could be a leap year if
                                        % it's divisible by 4
        check_centennial(Year).

    check_centennial(Year) :-
        0 <> Year mod 100, !.           % year is divisible by 4
                                        % but is not a centennial

    check_centennial(Year) :-
        0 = Year mod 400.
```

Evaluating Expressions

A mathematical expression in Prolog can be any number of numeric values separated by arithmetical operators. A string of mathematical expressions to be evaluated must not contain any free variables because then a precise value cannot possibly be obtained if some of the values to be evaluated are unknown. For example, the following expression can be fully evaluated because all of the arguments involved have set numerical values:

$3 * (-4.4 / 2.3 + 5.93E+23)$

When evaluating numerical expressions it is not good enough merely to calculate a particular value. For the evaluation to be of any use, you must either make a comparison with the value obtained or pass the calculated value along to where it can be used.

In Prolog a full set of relational operators is provided to aid in comparing values. With the use of the equality predicate, you may bind a variable to a particular value or Prolog term. That variable can then be passed from call to call.

Prolog's Relational Operators

Table 8-2 lists the various relational operators that are used to make comparisons in Turbo Prolog. All of these operators use infix notation, meaning that they must be set between two numeric values before a comparison can be made. The numeric values on both sides of a comparison must be able to evaluate to precise numerical values. No free variables are allowed on either side of the comparison operator.

The Equality Predicate

You may have noticed that the equal sign (=) is included in Table 8-2. This is because Turbo Prolog uses the equal sign for two completely different tasks. Most procedural programmers know the equal sign well, using it to assign particular values to variables. However, in Prolog the equal sign does not make an assignment, as in other programming languages. Using the equal sign, values may be compared with other values or a Prolog term can be made to match another Prolog term.

Symbol	Meaning
<	Less than
>	Greater than
=	Equal to
<=	Less than or equal to
>=	Greater than or equal to
<> or ><	Not equal to

Tabla 8.2 *Relational Operators*

The equal sign can be used as a *comparison operator*, comparing the left side of an equation with the right side. The operator makes a logical check to see if the terms on both sides of the comparison are equal. If both sides prove to be equal, processing may continue forward. As with the other relational operators, both sides of the comparison must contain expressions that can be fully evaluated; thus, no free variables are allowed.

The equal sign is also a predicate that is used with infix notation. The *equality predicate* will succeed if the objects on both sides of the equal sign can be made to match. Instead of asking if two things are equal, as with the equal comparison operator, the equal predicate tries to make two objects equal through unification. For example, take the expression

$$Z = 3 + 4$$

If Z is bound when this line of code is called, then the equal comparison operator will be used because both sides of the equal sign contain values that can be evaluated. The outcome will be either success or failure, depending on the comparison of the two values.

If Z is free when this expression is encountered, the equality predicate will be called and the variable Z will become bound to the value that the equation on the right evaluates to. When the equality predicate is called with numeric values, only one side may contain a free variable. One side of the expression must be able to evaluate to a complete value; otherwise an error message will state that the expression contains a free variable.

Incrementing Values

Many procedural programmers shudder at the idea of learning how to program without using assignment statements. This habit is hard to break if you are coming to Prolog from a nondeclarative language. According to the discussion on the equal predicate, this line of code makes no sense in Prolog:

$$X = X + 1$$

For this expression to be evaluated, X must be bound, and X can never equal $X + 1$!

Remember that once a variable becomes bound inside a clause, the only way to free that variable is through backtracking. There is no mechanism in Prolog that allows for a variable to change its value inside a clause once it is bound. Because of this a new variable must be created when you wish to increment a value. The following statement shows how this can be done:

$X1 = X + 1$

Here the variable $X1$ holds the incremented value. Although at first this may seem like a nuisance, incrementing values in this fashion is easy to get used to. After a while you'll be amazed to see that it makes sense to increment values in this way.

Mathematical Functions

In addition to infix operators, Turbo Prolog also provides a generous list of miscellaneous mathematical functions, including a full range of functions used to evaluate logarithmic and trigonometric equations. Each operand in Table 8-3 uses the standard Prolog format of placing the relationship in front of its list of arguments, as in

relationship (object)

The use of prefix notation allows the functions to operate on a single value, returning a result to the value supplied. The outcome of the evaluation may either be compared to another value, used in a larger mathematical equation, or set equal to a variable using the equal predicate. For example, the following code shows how the free variable Z becomes bound to the absolute value of -33.33 :

$Z = \text{abs}(-33.33)$

Using the same function, the absolute value of a number may be used in the body of a mathematical expression:

$Z = 1.33414E-5 / \text{abs}(X)$

In this instance, the variable X must be bound prior to making a call to this statement.

Radians or Degrees

When using the trigonometric functions supplied with Turbo Prolog, be sure to remember that the functions evaluate their arguments in units of radians and not degrees. This is done because it is easier to calculate graphics coordinates with radians (a common use of trigonometric functions). If you want to convert from degrees to radians, the process is simple. The relationship of radians to degrees is

$\text{Degrees} = \text{Radians} * (180 / 3.141592653)$

Miscellaneous

$\text{abs}(X)$	Returns the absolute value of X (if X is positive, $\text{abs}()$ returns X , otherwise, $\text{abs}()$ returns $X * -1$)
$\text{round}(X)$	Returns the rounded integer value of X
$\text{sqrt}(X)$	Returns the square root of X
$\text{trunc}(X)$	Returns the truncated integer value of X

Logarithmic

$\text{exp}(X)$	Returns "e" raised to the power of X
$\text{log}(X)$	Returns logarithm of X in base 10
$\text{ln}(X)$	Returns logarithm of X in base "e"

Trigonometric

$\text{arctan}(X)$	Returns arctangent of X
$\text{cos}(X)$	Returns cosine of X
$\text{sin}(X)$	Returns sine of X
$\text{tan}(X)$	Returns tangent of X

NOTE: All trigonometric functions require that X be in units of Radians.

Table 8-3. Standard Mathematical Functions

Knowing this, you may now build a conversion formula to supply the proper units to the trigonometric functions. If you want to use a trigonometric function, but only know the degrees, you can easily convert the degrees to radians using this formula:

$$\text{Radians} = \text{Degrees} / (180 / 3.141592653)$$

For example, if you want to find the tangent of 45 degrees, you can use the following clause:

```
tangent_45_degrees(Tangent) :-
    Rad = 45 / (180 / 3.141592653) ,
    Tangent = tan(Rad).
```

This code can be generalized to find the tangent of any degree you wish to supply:

```
tangent(Degrees, Tangent) :-
    Rad = Degrees / (180 / 3.141592653) ,
    Tangent = tan(Rad).
```

Rounding and Truncating

Two built-in functions allow for the conversion of real numbers to integer values. You may use either one, depending on how you want the conversion from real to integer to take place. The function

`round()`

rounds a real number to its nearest integer value. Numbers that have exactly half an integer value will always round up to the next integer. For example, the following statement is true:

`5 = round(4.5)`

On the other hand, the function

`trunc()`

drops off any fractional part of a real number to create an integer value. In this case, the following statement is true:

`4 = trunc(4.5)`

Adding a Random Flare

Turbo Prolog provides a built-in predicate that allows you to create a random sequence of numbers in your programs. Random numbers are especially useful when you are testing algorithms or are providing a “what if” feature to your programs. When using `random()`, there is no need to compare or set the result to a value, since the actual response to the call is returned as an argument to the predicate.

There are two different randomizing predicates. The first one,

`random(X)`

makes use of a single argument. When this version is called, the argument must be a free variable, and a random number between zero and one is returned bound to the variable. Notice that this version of `random()` can return a value that is equal to zero. Thus, the result from a call to `random()` can be represented by the following:

`random(Z)` where $0 \leq Z < 1$

This simply states that when `random()` is called with a single argument, the result will be a real number greater than or equal to zero, but less than one.

The second form of `random()` takes two arguments and returns a random integer value when called:

`random(MAX, X)`

In this version, the first argument specifies the maximum integer value to be returned, while the second argument supplied must be a

free variable. When the two-arity version of `random()` is used, the return value is represented with this diagram:

`random(MAX, Z)` where $0 \leq Z < \text{MAX}$

In other words, the integer that `random()` returns will be greater than or equal to zero and less than the maximum number specified.

Modeling Your Own Predicates

Turbo Prolog can't supply all of the predicates you may need to complete a certain task. For example, if you look closely, you'll notice that a function for calculating the power of a number is not given. This shouldn't pose a problem, though, because it is easy enough to model a power function of your own using the tools provided.

One of the beauties of programming in Prolog is that if a function or predicate doesn't exist you can build it yourself. With the power function it is easy to see how to model a power of 2. Simply multiply a number by itself:

`X = Number * Number`

Here, X is bound to the value of a number raised to the power of 2. The power of 3 is also easily modeled:

`X = Number * Number * Number`

However, at this point it is clear that an actual function would be better than writing out each version of power that may be needed. Imagine writing out a function for a power of 50 or more. It should be easier to look up a formula for calculating a power than to write and debug a power of 50 predicate using the method shown above. A formula for calculating the power of a number is

$M^n = \exp(n * \ln(M))$

This formula states that to raise M to the n th power, you take the $\exp()$ of n multiplied by the natural log of M . To incorporate this into a Turbo Prolog predicate, use the following code:

```
Predicates
  power(real, real, real)
Clauses
  power(M, N, Result) :-
    Result = exp( N * ln(M) ).
```

You now have created a predicate that can be used to calculate exponents. To calculate the value of 45^{17} , simply call `power(45, 17, Result)`.

Comparing Prolog Terms

Because numbers in Prolog are treated the same as other symbols, you can use the relational operators described in this chapter to compare more than just numerical values. Except for compound terms and lists, any Prolog term may be compared with the use of the relational operators. For example, when writing a sorting routine, the greater-than operator can be used to make comparisons of any type, whether string, character, or symbol.

When comparing strings, characters in the strings are compared one at a time. The comparison succeeds or fails depending on the outcome of each comparison. If a comparison fails while comparing successive characters, the entire comparison will fail.

Symbols, like strings, can be compared for equality, nonequality, and less-than or greater-than. When two different symbols are compared they can either be bound to variables (which are then passed to the comparison) or compared directly. If the symbols are directly compared the symbols themselves must be enclosed in double quotes, as if they were strings.

Comparing characters is no different than comparing strings, except that the characters must be enclosed in single quotes if they are compared directly. To increase the speed of processing, Turbo Prolog converts all characters to their respective ASCII integer values.

Because of this conversion, all character and string comparisons are actually made at an integer level.

Comparing Compound Objects

While it is easy to compare most Prolog objects with built-in operators, sometimes you may want to see if two compound objects are equal, or you may wish to unify two Prolog terms. To do so requires that you build a predicate that can make this comparison. The predicate below is so simple you may wonder why you didn't think of it yourself:

```
equal(X,X)
```

The only requirement for this predicate is that you declare it with the proper domains that you want to use for the comparisons. Here is a small program that shows how this predicate can be put to use:

```
Domains
    name      = n(first, last)
    first, last = string
    age       = integer

Predicates
    client(name, age)
    equal(name, name)
    find_duplicate_names

Clauses
    equal(X,X).

    client( n("Sally", "Smith"), 54).
    client( n("Barbara", "Johnson"), 36).
    client( n("Ruddy", "Naples"), 27).
    client( n("Kenny", "Batherton"), 57).
    client( n("Barbara", "Johnson"), 19).

    find_duplicate_names :-
        client(Name_1, Age_1) ,
        client(Name_2, Age_2) ,
        equal(Name_1, Name_2) ,
        Age_1 <> Age_2 ,
        Name_1 = n(First, Last) ,
        write("Duplicate name found: " ,
            First, " ", Last, " Ages: " ,
            Age_1, " and " , Age_2, ".\n"),
```



```
        fail.  
    find_duplicate_names.  
  
GOAL  
    makewindow(1,2,3," Duplicate Names ",0,0,25,80) ,  
    find_duplicate_names.
```

Type Conversion

When comparing two values, the domains of the values must be of equal type. While some conversions are handled automatically by the system; other conversions must be made specifically. A few built-in predicates are handy for converting data from one type to another:

```
str__char()  
str__int()  
str__real()
```

While Turbo Prolog version 2.0 automatically converts characters to integers as needed, it is sometimes required to convert strings to characters, reals, or integers. Notice that all of the conversion predicates can go either way; a string may be converted to a real number, or a real number to a string.

When converting from a string to another data type, the string value must be able to be represented as the new data type. If not, the conversion will fail. For example, the string "Sammy" cannot be converted to an integer, a character, or a real number.

Another useful conversion predicate is

```
upper__lower()
```

This predicate can be used with either strings or characters and converts the input value to its respective upper- or lowercase, depending on how the predicate is called. When you want to make a comparison and do not know if the original value is in upper- or lowercase, `upper__lower()` is extremely useful. The following short routine shows a way to ask for input and to test if "yes" was input in either upper- or lowercase:

```
test_input :-  
    write("Do you want to continue? (Yes/No): ") ,  
    readln(Ans) ,  
    upper_lower(Ans, "yes").
```

This routine will succeed as long as you enter **yes** in any case of characters. All uppercase or lowercase letters will be converted to lowercase, and the answer will be compared with the string "yes". If any response other than the word yes is given, the predicate will fail.

Hex Notation

One last useful conversion is a form of numeric notation. In Turbo Prolog if you wish to represent an integer value as a hexadecimal number (hex for short), you may precede the numeric value with a dollar sign (\$). If hexadecimal notation is new to you, don't worry. It is usually used in advanced programming applications where programs are needed to interface directly with hardware devices. For more information on hex notation and its uses, refer to an advanced manual on DOS.

Part Three

Up to now this book has discussed the basics of using Turbo Prolog. This section will introduce some of the program's unique features that will enable you to write more sophisticated and powerful programs. These advanced topics include advanced compiler features, recursion, and advanced programming techniques.

9 *Working with Lists and Recursion*

Often, when defining relationships in Prolog, it is easy to tell how many arguments a predicate will require. For example, when writing a set of facts to describe the parts of a bicycle, it is not difficult to determine the number of arguments that are needed to represent each particular section of the bike. The entire bicycle can be broken down into wheels, a frame, the drive train, a seat, brakes, and handle bars. In turn, the wheels can be broken down into the tires, rims, spokes, and hubs. A set of facts that describes a bike might look like this:

```
bicycle(frame, wheels, drive_train, seat,  
        handle_bars, brakes).  
wheels(tire, rims, spokes, hubs).  
drive_train(chain, cluster, crank, sprockets, pedals).  
frame(headset, fork, body).  
seat(seat_post, saddle).  
handle_bars(neck, bars).  
brakes(handles, cords, brake_pads, brake).
```


One drawback of representing each bicycle part in this manner is that an entirely different predicate is needed to describe each component to be inventoried. Also, a separate rule must be written to process each set of parts because there is no one fact that generally describes the different pieces of the database.

Another way of representing this same information is to use compound objects. Using this scheme of data representation, it is not necessary to declare a new predicate for each new major component. Instead, a new compound object is declared for each set of subparts. By using compound objects processing the parts can be greatly simplified:

Domains

```
major_part = symbol
```

```
sub_parts = p(symbol, symbol) ;
            p(symbol, symbol, symbol) ;
            p(symbol, symbol, symbol, symbol) ;
            p(symbol, symbol, symbol, symbol, symbol) ;
            p(symbol, symbol, symbol, symbol, symbol, symbol)
```

Predicates

```
bicycle_part(major_part, sub_parts)
```

Clauses

```
bicycle_part(bike,
              p(frame, wheels, drive_train, brakes,
                 seat, handleBars)).
bicycle_part(frame, p(headset, fork, body)).
bicycle_part(wheels, p(tire, rim, spokes, hub)).
bicycle_parts(drive_train,
              p(chain, cluster, crank, pedals, bottom_bracket)).
bicycle_part(brakes,
              p(handles, cables, brake_pads, brakes)).
bicycle_part(handleBars, p(neck, bars)).
```

But this method soon gets as tedious as the first solution. Not only must you keep adding new compound declarations as you go along, but also you must write separate procedures to handle each major component. Because no two sets of subparts can be guaranteed to contain the same number of items, no general rule can be written to take care of all the possible sets of subparts.

The List Data Structure

In the bicycle database, each major component may contain a unique number of subparts. What is really needed to handle data is a structure that may contain an unspecified number of values. The list structure becomes an ideal data type to represent the subparts in this program because with lists there is no need to declare the number of objects an argument may contain.

Like the compound objects introduced in Chapter 6, lists can be used to pass several values with the ease of using a single argument. Although lists and compound objects have this likeness, the similarity ends here. Lists, unlike compound objects, have the advantage of being able to hold from zero to many values, and neither you nor the compiler need to know in advance how many items the list may contain.

Declaring Lists

Lists are complex data structures that have no corresponding system-defined domain type. This means that a special domain must be specifically declared for each type of list you want to use in your program. Normally, the elements in a list will all be from a single domain type, such as with the following list declaration:

```
Domains  
  integerlist = integer*
```

This declaration is straightforward; it declares a domain to represent a list of integers. (If you have programmed in C before, be sure not to confuse the asterisk in the list declaration with the use of the asterisk in a C program. C uses the asterisk to indicate a "pointer" value. In Turbo Prolog, there is no feature that allows you to reference values in this fashion.)

List Notation

You may have already seen the notation used for lists: an open square bracket followed by zero or more *elements*, all followed with a closing square bracket. In lists, each element (or value) in the list is separated by a comma. With this notation, an integer list, as is declared above, can take on the value of

```
[1,2,3,4,5,6,7,8,9,0]
```

This entire series of values can now be passed and manipulated as a single argument. Here are some more examples of lists that can belong to the domain of "integerlist":

```
[8,20,58]
```

```
[1,1,1,1,1,1,0,1]
```

```
[]
```

Using lists the bicycle database can now be rewritten to be quite manageable. With this data representation, only one predicate is needed to hold all the subparts. The program now looks less complex and is easier to maintain:

Domains

```
sub_parts = symbol*      % a list of symbols
major_component = symbol
```

Predicates

```
bicycle_part(major_component, sub_parts)
```

Clauses

```
bicycle_part(bike, [frame, wheels, drive_train,
                    seat, handle_bars, brakes]).
bicycle_part(wheels, [tire, rims, spokes, hubs]).
bicycle_part(drive_train, [chain, cluster,
                          crank, sprockets, pedals]).
bicycle_part(frame, [headset, fork, body]).
bicycle_part(seat, [seat_post, saddle]).
bicycle_part(handle_bars, [neck, bars]).
bicycle_part(brakes, [handles, cords, brake_pads, brake]).
```

With this representation no new declarations need to be made as you add new components to the bicycle. If a new major component is needed, it is easy enough to add the appropriate fact describing

the part and its subparts. Because one predicate describes all the different bicycle parts, generalized routines can be written to manipulate the database. As new parts are added to the database no new predicates need to be written because all the parts are represented within the same data scheme.

The Empty List A list has the flexibility of being able to contain any number of elements. Not only this, but a list can also be represented as being empty. The *empty* (or *null*) list is shown with a pair of brackets, without any list elements in between:

[]

The empty list is a special data item in Prolog. Although not normally seen, each list in Prolog is actually terminated by an empty list. When writing list-handling procedures, the empty list is often used to test for when the end of a list has been reached.

In Prolog, a special list is the *singleton list*, or a list with a single element in it. Written as

[element]

the singleton list contains an extra, hidden element: the empty list. This is important to realize when working with lists and when writing list-handling predicates.

Head/Tail Notation To simplify list processing a list can be broken down into two parts: a head and a tail. The *head* of a list is always the first element of a list, while the *tail* is the remainder of the list without the head. The head of a list always contains a value from the domain that the list is made up of. Since the tail of a list is always a list itself, it can either contain values or be empty.

In list notation, the head of the list is separated from the tail with a vertical bar, like this:

[Head | Tail]

Actually, this definition of list structure is slightly misleading, because the head of a list can also represent more than a single list

element. This is done by placing more than one element in the head before separating it from the tail. Here is an example of how several values can be pulled from a list:

```
[ Head_1, Head_2, Head_3 | Tail ]
```

The variables Head_1, Head_2, and Head_3 each become bound to a single list element, while Tail is the same list of elements, but without the first three elements. As always, the tail is still a list.

Unification and Lists

Binding a variable to a value takes place when a free variable can be made to match with a value. The process that binds the variable to a value is called *instantiation*. A variable may either be bound to a value (instantiated) or it may be free (uninstantiated). Unification and pattern matching go hand in hand; one cannot exist without help from the other. While a fine line exists between instantiating a free variable and unification, the difference can easily be seen when two lists are unified.

Unification allows a complete match to take place between two terms. Unification can take place between two free variables, two terms, or a free variable and a term. Table 9-1 shows some examples of the unification between two lists.

With most of the examples in Table 9-1 it is hard to see the difference between the binding of variables and the unification of terms. However, the difference between unification and the instantiation of variables can be seen in the last example, where the list [a,A,Y] is unified with the list [X,1,X]. When two terms are unified, all like variables must be able to represent the same value; otherwise the unification will fail. Here, the variable X first matches with the value a, causing all instances of the variable X to take on the value a. When this happens, Y becomes bound to a, because Y is matched with X. The unification returns with both X and Y instantiated to the value a and A instantiated to 1.

List 1	Unifies With	List 2	Results
[1,2,3]	=	[A,B,C]	A = 1, B = 2, C = 3
[1,2,3,4]	=	[A,B C]	A = 1, B = 2, C = [3,4]
[tom, X, rebal]	=	[Y, ellis, Z]	Y = tom, X = ellis, Z = rebal
[a]	=	[H T]	H = a T = []
[a,b,c]	=	[X,Y,Z Tail]	X = a, Y = b, Z = c, Tail = []
[a [b [c []]]]	=	X	X = [a,b,c]
[]	=	[H T]	Fails to unify
[a,A,Y]	=	[X,1,X]	A = 1, Y = a, X = a

Table 9-1. Unifying Lists

If, on the other hand, Y was already instantiated to a value when these two terms attempted to unify, a comparison would be made between the value a and the value that Y was bound to. If the two values were not equal, then unifying the terms would not succeed.

Recursion

The word recursion stems from the Latin word *recurrere*, which means "to run back." In programming languages, something that is recursive is said to be something that is defined in terms of itself. Due to the way in which lists are structured, recursion becomes the natural way to process these data objects.

Problem Reduction

A recursively defined task is a process in which the task is simplified by one step before the whole process is called again. Since each step of recursion takes you one step closer to the goal, you will eventually reach the last step, which will land you at the end of the task. At this point, you will have satisfied the goal and the processing can stop. This approach to solving a problem is known as *problem reduction*, and is often used in everyday routines.

The general form of recursion can apply directly to processing lists. Here is an example:

```
do_task( task = done ).           /* goal state */
do_task( task ) :-
    process( part_of_task ),      /* reduction */
    do_task( rest_of_task ).      /* recursion */
```

The first clause is called the *terminating condition*, because it checks to see if processing has reached the goal state. If it has, processing can stop and a return can be made to the original call.

The second clause breaks off a part of the task to process and

takes care of that part, which reduces the overall work to perform. Once a portion of the task has been taken care of, a call is made to start the process over again, this time with the reduced portion of the task. At the beginning of each step, the terminating condition is checked to see if the task has been completed. If the task is not completed, then the problem reduction process starts over.

Recursion in List Processing

A classic example of a list-handling predicate is `member()`, where a list is tested to see if it contains a specific element. The predicate `member()` is defined with two clauses—a terminating condition and a clause that reduces the task.

When defining a list-handling routine, it is easiest to start with the terminating condition. With `member()`, the goal state is an easy one; if the value in question is at the head of a list, then you have reached the stopping point because the element has been found. This goal state can be written as

```
member(Element, [Element|_]).
```

Notice that in this clause an anonymous variable is used to represent the tail of the list. In this clause the `rest_of_task` is not important because the value being searched for has been found. The remaining part of the list is no longer needed.

The second clause, which defines `member()`, contains the recursive call

```
member(Element, [_|Tail]) :- member(Element, Tail).
```

The anonymous variable is used in this clause because the head of the list has proven not to match the value being looked for, and instantiating the head of the list with a variable name would only be a waste of computer resources, and the compiler would make a point of this by warning that “the variable is only used once.”

Normally, the “variable is only used once” warning message is given to help check for errors with the spelling of variable names. If a variable is only used once inside a clause, then there is no need to use a variable name at all. If this is the case, use the anonymous variable to represent the argument at the position in question. Because you will not be passing on the value that would become bound to such a variable, you do not need to use a name to represent the value that the argument represents.

The two clauses placed together (and slightly shortened) show the full definition of `member()`. The necessary domains and predicates declarations have also been added to give this program:

```
Domains
    symbol_list = symbol*

Predicates
    member( symbol, symbol_list)

Clauses
    member(X, [X|_]).
    member(X, [_|T]) :- member(X, T).
```

While not exactly like the general form of recursion, `member()` comes close to being the ideal model for a recursive predicate. The difference is that `member()` does not do any actual processing at each pass through the recursive process. The predicate simply steps through each element of a list, testing for equality at each new pass.

Stepping through the list is accomplished by successively calling `member()`, each time reducing the original task by leaving off the head of the original list. While this may not seem intuitive at first, a close inspection of the processing will show how this predicate works.

Tracing Recursive Predicates

Giving the goal `member(p, [h,a,p,p,y])` to the program above will result in “Yes” being returned to the Dialog window. Following this call in the Trace window, the first clause of `member()` quickly fails because the first element in the list does not match the value being sought. In turn, the second clause is tried, splitting the list

up into its proper head and tail values. Doing so causes the tail of the list (the variable *T*) to become bound to the list *[a,p,p,y]*. Processing continues on with a recursive call to *member()*. Since the tail of the list is a list itself, it can be supplied to the second argument of *member()*. The recursive call now looks like this:

```
member(p, [a,p,p,y])
```

Following the trace, you can see how the first clause of *member()* fails in its attempt to match with the call. But now a curious thing happens. The call *member(p, [a,p,p,y])* does match with the second clause of *member()*. You may think that this is strange, since you know the variable *T* is already bound to *[a,p,p,y]*. How can the first element of this list be taken off and then made to match with *T*? The answer is that Prolog is processing an entirely new call to *member()*. The variables in this call have no relationship to the variable bindings contained in the previous call to *member()*. Because of this, the variable *T* is not instantiated to a value, and the call matches by unifying with the list *[a,p,p,y]* using the construct *[_|T]*. In doing so, the variable *T* becomes bound to the list *[p,p,y]*.

Once again, another call to *member()* is made, this time with the value *p* and the list *[p,p,y]*. Here, the call succeeds on the first clause of *member()* and the original call returns true.

While watching the series of calls and returns in the Trace window, you will notice that for each recursive call made to *member()* a return is made. When processing recursive clauses, Prolog keeps track of each call because Prolog must successfully return from each call to satisfy the goal. When Prolog winds down into a recursive procedure, it must fully unwind to prove a query true.

Using Unification to Test

In the first clause defining *member()*, unification is used to test the head of the list against the value being searched for. A more indirect method can be used in this clause to test for equality:

```
member (Element, [Head|_]) :- Element = Head.
```


While this coding adds redundancy to the definition, it brings to light an interesting property of `member()`. It will act in different ways, depending on which values are bound and which are free when called. When both the first and second arguments in the call are bound, a comparison is made between a value and the head of the list. The clause succeeds if the two are equal. On the other hand, if the first argument is free when called, the variable will become instantiated with the value in the head of the list. Even though both cases shown for the first clause of `member()` are really the same, the power of unification allows for the entire clause to be written as a fact, thus shortening the processing time needed and clarifying the code.

Winding Down into Recursion

With the aid of backtracking, `member()` can show another property of recursion. Using the call given below, `member()` will return all the elements of a list, one at a time:

```
member (Element, [n,e,w]).
```

With this goal, Turbo Prolog will output

```
Element = n  
Element = e  
Element = w  
3 Solutions
```

The recursive property of `member()` allows for the return of all elements in the list. At first this may seem surprising, but taking a closer look will reveal why this works.

When the call is first made, the first clause defining `member()` succeeds and returns the first solution. However, before returning, Prolog notices that there is another possible path that may also solve the goal. By forcing backtracking, `member()` can be made to return yet another solution.

While tracing this call, the Trace window will output the line

```
REDO: member (_,["n","e","w"])
```

indicating that Prolog is looking for another solution to the call (See Figure 9-1). The Trace window will always represent free variables as underscores, and all symbols in the trace will be enclosed in double quotes.

When the call to `member()` is redone, Prolog drops down and tries the second clause defining `member()`. Here, the first element is stripped off the list, and `member()` is called again, this time with the tail of the original list. The new call made is

```
member(Element, [e,w])
```

At this point, Prolog makes an entirely new call to `member()`,

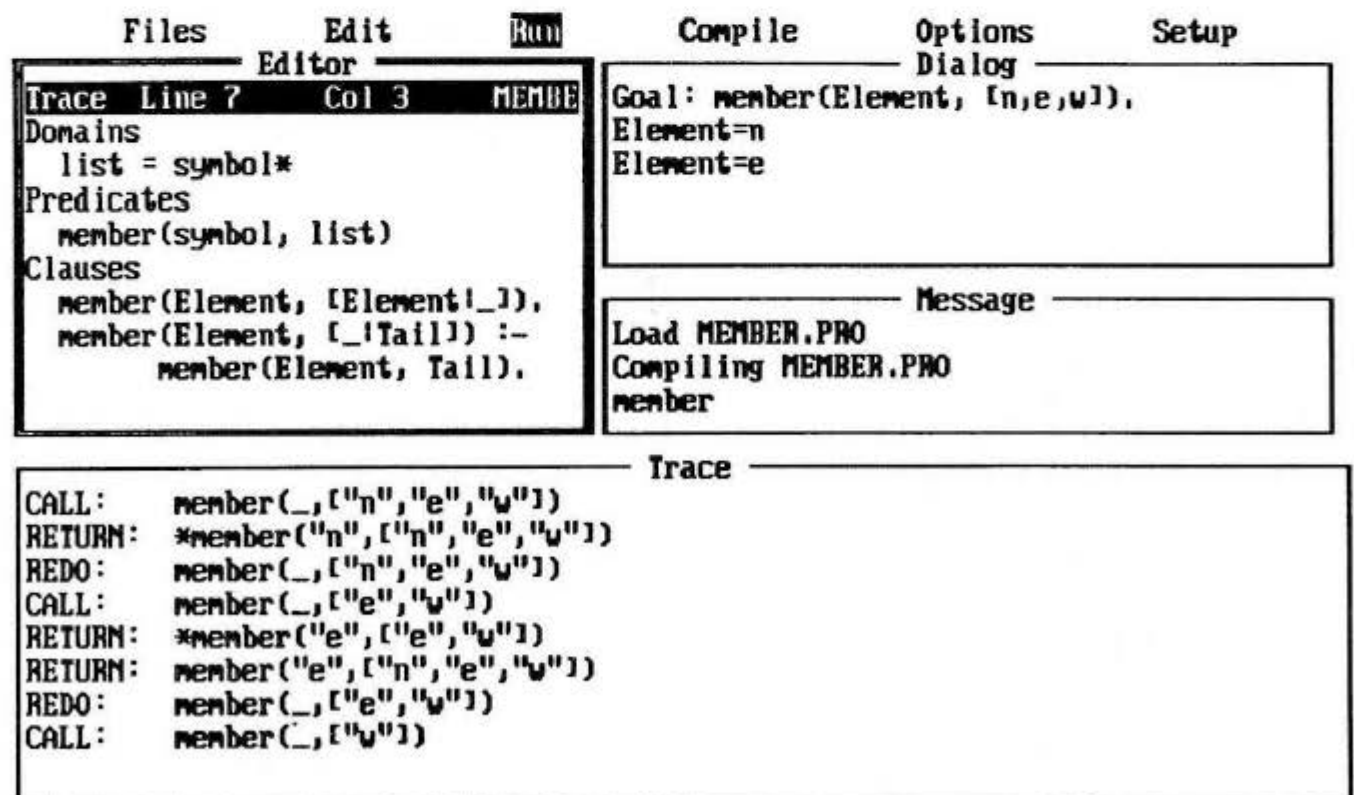


Figure 9-1. Tracing a recursive predicate

looking for alternate solutions to the original call. In order for Prolog to find another solution, this new call must succeed, which it easily does by finding a match with the first clause of `member()`. The next solution can now be returned (the symbol `e`), but in doing so, Prolog looks ahead and sees another possible solution to this latest call to `member()`. After Prolog returns the value `e`, backtracking forces Prolog to redo the call `member (Element, [e,w])`.

When backtracking redoes this call, the second rule of `member()` is encountered again. The head of the list is stripped off, and a recursive call is made with the remaining tail of the list. As before, two possible solutions exist for this new call. The first solution returns the head of the list, while the second solution again makes a recursive call with the tail of the list.

Following the trace through these calls, you will see that when the last recursive call is made, the list in the second argument to `member()` is an empty list. Since the empty list fails at matching with any of the clauses defining `member()`, the last call ends up in failure.

This example has been presented in detail to point out that when winding down into a recursive procedure, each new call to `member()` invokes a brand new call. With each new call made, all the opportunities exist for that call either to succeed or fail. Values can be returned and backtracking points set with each new call that is made in a recursive process.

Assigning Values in a Recursive Process

Although a call into a recursive predicate may take Prolog through many levels before a solution is found, Prolog will always return to the call with the same arguments that were originally presented. Input arguments will always return with the same values they had when the call was made, and free variables in the call have the capability of returning values gained in processing.

Passing values out of recursion can be seen in the example just described, where `member()` is used to return all the values in a list.

When tracing the example, you may have noticed that each time a return was made, Prolog had to step back (return) through each of the recursive calls before a value could be fully returned and printed out. Each level of recursion must be returned through, and the value found has to be carried all the way back to the original call.

In the End a Solution Is Found

In recursion, the values that are returned are usually found at the bottom of the recursive cycle (why keep looking for something once you have found it?). Because of this, it is good practice to instantiate return variables in the terminating condition of the procedure. The procedure that finds the length of a list is a good example of how a value can be passed from the terminating condition of a recursive procedure.

Domains

```
integerlist = integer*
counter = integer
```

Predicates

```
list_length ( integerlist, counter, counter )
```

Clauses

```
list_length([], Length, Length).
list_length([H|T], Length_in, Length_out) :-
    New_length = Length_in + 1,
    list_length(T, New_length, Length_out).
```

The procedure `list_length()` is made up of two clauses. This procedure will calculate the length of a list provided and return the length as an integer. The second clause does most of the processing by taking off the first element of the list and adding one to a counter. This works because after each element of the list is counted, the counter is passed over to the output argument and the procedure returns out of the recursion. The passing of the counter is done in the terminating condition of the procedure:

```
list_length([], Length, Length).
```


This clause effectively describes a condition for when the input list is empty. When encountering an empty list, this clause will pass the counter (contained in the second argument) to the output argument of the predicate. This predicate works because when first called, the counter value is initialized to zero, like this:

```
list_length([], 0, Length)
```

Here you can see that the terminating clause will return the correct solution given the situation of an empty list. The correct counter value is passed to the third argument, where it can return out. Now, all that is needed to have an effective list counter is to keep track of each item in the list.

The second clause works by adding one to the current counter value, then passing on this value as the new counter value. With each level of recursion, the task is simplified by taking one element off the head of the list and adding one to the counter, until the end of the list is reached. Compile and run `list_length()` in Trace mode, giving the goal

```
list_length([2,4,6,7,3], 0, Length).
```

When this call makes its final return in the Trace window, note that the first two arguments are equal to the arguments first supplied to the call (the list `[2,4,6,7,3]` and 0), and the third argument holds the value equal to the number of items that are in the list.

Creating Lists

Recursive procedures can be written to create new lists. When this is done, a list is constructed by adding elements, one at a time, to the head of a list. In other recursive procedures, the initial return value is found at the bottom of the recursive cycle and making new lists is no exception. Creating a list is accomplished by *initializing* the return argument to the empty list in the terminating condition. After the list is initialized, elements can then easily be added on to the head of the list as the procedure unwinds from recursion.

The predicate `remove_dups()` provides a good example of how a list can be created through recursion. The procedure `remove_dups()` takes two arguments: an input list, and a list with all of the duplicate elements removed.

Removing the duplicated elements is accomplished by stepping through the original list. After this, the second list is created when stepping out of recursion by adding only the elements that have not yet been added. The effect is to create a list in which all the duplicates have been removed.

Initializing the New List

It is easiest to write the procedure by starting with the terminating clause. In `remove_dups()`, this clause serves a dual purpose. First, it satisfies the condition for when an empty list is supplied to the predicate. Second, and more importantly, the clause initializes the return list, making it easy to build up the new list from the terminating condition. When writing this clause, it is easy to see why Prolog is known as a descriptive language—if the first list is empty, a list with all the duplicates removed will be empty too:

```
remove_dups([], []).
```

List-creating predicates builds up the return list while unwinding from the terminating condition. Because of this, it is common to initialize the return list to a null list, since every list must end this way. Here is the full definition of `remove_dups()`:

Domains

```
list = symbol*
```

Predicates

```
remove_dups ( list, list )
member ( symbol, list )
```

Clauses

```
remove_dups([], []) :- !.
remove_dups([H|T], T1) :-
    member(H, T), !,
    remove_dups(T, T1).
remove_dups([H|T], [H|T1]) :-
    remove_dups(T, T1).
```

A quick look at this definition shows that it takes more than two clauses to define the procedure `remove_dups()`. Not only is there a terminating clause, but also the situations must be taken into account for when the element in question is a duplicate and when it is not.

If the head of a list is found to be a member of the tail of that same list, then the element is duplicated inside the list. Given this reasoning, the second clause of `remove_dups()` uses `member()` to test for duplicate elements. This clause succeeds when a duplicate element is encountered. Since duplicate items are not added to the new list, the return list is left as is. The clause ends by making a recursive call with the tail of the first list. This kind of processing can be seen in this definition:

```
remove_dups([H|T], T1) :-
    member(H, T), !,
    remove_dups(T, T1).
```

Adding Elements to the List

The last clause in `remove_dups()` is the clause that actually builds up the return list. If the head of the first list is not a duplicate element (the second clause has failed the `member()` test), the element will be added to the head of the return list:

```
remove_dups([H|T], [H|T1]) :-
    remove_dups(T, T1).
```

The last two clauses defining `remove_dups()` both reduce the problem by stepping through the first list one element at a time, and make the recursive call with a smaller input list. Duplicate elements are not added to the new list, while the elements that are not duplicated are tacked onto the head of the return list.

The cuts play an important role in the definition of `remove_dups()`. Without the cuts, it would be possible for Prolog to generate more than one solution to a call, and the list could return with duplicated elements. With the cuts in place, the definition assures that only one clause can succeed for any given call to `remove_dups()` (remember the discussion on case statements in Chapter 7).

Building Lists Dynamically

While recursive predicates such as `remove_dups()` can be used to modify an input list, often you will wish to create a list from scratch. The most straightforward way to build and take lists apart is with the predicate `make_list()`:

```
Domains
    list = symbol*

Predicates
    make_list(symbol, list, list)

Clauses
    make_list(Element, List, [Element|List]).
```

Type this program, testing it with the following queries:

```
make_list(dog, [cat, hen], New_List).
make_list(Element, New_List, [printer, system_unit, monitor]).
make_list(table, [desk, chair], [H1, H2 | Tail]).
```

While `make_list()` will add a single element onto the head of a list (or take the first element off) its use is limited because it must be placed inside a recursive loop if anything more than this needs to be done.

Depending on where the list items that you wish to collect are coming from, different techniques can be used. For instance, if you want to collect a list from a set of data that is already stored in your program, by far the easiest way of doing so is with the built-in predicate `findall()`. On the other hand, if the list elements are coming from user input or from a file, you will need to build a predicate that can create a list from the input.

The built-in predicate `findall()` can collect a list from data that is already contained in your program. The predicate has the capability to select the elements that are to be added to the list, based on a set of conditions that you specify. To use `findall()`, you must first declare a list domain of the type of list you wish to create. Using three arguments, `findall()` first takes a free variable, followed by a functor that has been declared as a predicate in the predicates section of your program. The third argument must also be a free variable, and is the argument where the list is returned.

The following program shows how `findall()` is used to collect a list of all the people who have a customer code of 15:

```
Domains
    customers = name*
    code = integer
    name = n(first, last)
    first, last = symbol

Predicates
    customer(name, code)

Clauses
    customer(n(kelly, kliener), 12).
    customer(n(brian, walker), 15).
    customer(n(john, mock), 14).
    customer(n(lisa, pizara), 15).
    customer(n(tim, isom), 15).

Goal
    findall(X, customer(X, 15), Customer_list) ,
    write(Customer_list).
```

While `findall()` makes it easy to create a list from data that is already present in the program, dynamically creating a list from user input is a different story. In the program shown below, the model of recursion is changed a little. The recursion is terminated when the user inputs a pound sign (#). When this occurs, Prolog drops down to the second clause defining `get_input()`, and initializes the list in this clause. To watch this program in action, follow the processing through in Trace mode.

```
Domains
    stringlist = string*

Predicates
    get_input ( stringlist )

Clauses
    get_input([String|Tail]) :-
        write("Enter an item to add to the list.\n" ,
            "    (enter # to quit): ") ,
        readln(String) ,
        String <> "#", ! ,
        get_input(Tail).
    get_input([]).

GOAL
    makewindow(1,2,3,"",0,0,25,80) ,
    get_input(List) ,
    nl, write(List).
```


The Append Predicate

Yet another predicate that is capable of creating lists is the predicate `append()`. Taking three lists as arguments, `append()` is capable of splicing two lists together, much like what the built-in predicate `concat()` does with strings.

Domains

`list = integer*`

Predicates

`append(list, list, list)`

Clauses

`append([], List, List).`

`append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).`

This predicate can be called with virtually any flow pattern, providing a powerful way to build lists up and tear them apart. However, `append()` is normally used to append two lists together, and when used in this manner, is called with the first two arguments bound to lists. The predicate will then append the list in the first argument onto the beginning of the list in the second argument.

The predicate works by stepping down through the first list until the empty list is reached. Once this happens, the second list is moved into the third argument position, where new elements are successively added onto the head of the new list as the recursion unwinds.

In Prolog, creating and handling lists can be one of the more confusing elements of the language. Only time and practice will allow you to feel comfortable when it comes time to incorporate these data structures into your program. In the meantime, don't let the idea of recursive predicates scare you away from using lists in your program. Recursive procedures and lists can simplify your code and add elegance to the predicates you write.

10 *Input and Output with Turbo Prolog*

A programming language wouldn't be much good if it couldn't read data from a file or output results to a printer. When Prolog was originally designed, its sole purpose was to solve logical problems, so there wasn't much need for the language to contain strong file interactions. As the language grew in popularity, the scope of the language began to change into one that was used for an increasing number of tasks. With the introduction of Turbo Prolog, Prolog has blossomed into a flexible applications language capable of creating programs that utilize complex I/O and string-handling routines.

Working with Files

Files are used to store data—you either open a file to read some data in or you open a file to write some data out. The process should not be complicated, and with Turbo Prolog this process is made as simple as it sounds. The powerful file-handling predicates built into Turbo Prolog make writing to and reading from files almost transparent.

Redirecting the Streams

For a file to store data, a file is opened, read from or written to, then closed. Aside from these three actions, all that needs to be done is to tell the program where the data is going. In Prolog, input and output to the program travels through the streams of the program. There are two I/O streams in Prolog, one for input and one for output. By default, the input and output streams are the computer keyboard and monitor, respectively.

To change where the program reads its data from, the input stream must be changed. To tell the program to write data to a specified location, the output stream must be changed. This redirecting of the I/O is taken care of with the two predicates

`readdevice()`

and

`writedevic()`

Each redirection predicate takes a single argument consisting of a `SymbolicFileName`. You may redirect the input or output stream to any one of the predefined files shown in Table 10-1, but creating your own file requires that you declare your own `SymbolicFile-`

Predefined Files	Device
Screen	Computer monitor
Com1	Serial port
Printer	Parallel printer
Keyboard	Computer keyboard
Stdin	DOS standard input
Stdout	DOS standard output
Stderr	DOS standard error

Table 10-1. *System-Defined Symbolic File Names*

Name. This is accomplished by setting a name equal to the domain "file" in the Domains section of your program:

Domains

```
file = infile; outfile
```

Here, two SymbolicFileNames have been created for use in a program: infile and outfile. However, before the input or output stream can be changed, a file must first be opened for reading or writing, and its name must be assigned to one of the declared SymbolicFileNames. After doing so, you can then redirect an I/O stream to the SymbolicFileName used.

The program below shows how to open and write to a file using Turbo Prolog's file-handling predicates. Once the program is successfully compiled, you can run the program using the goal "open_for_writing."

```
/* Writing To a File */
Domains
    file = outfile

Predicates
    open_for_writing

Clauses
    open_for_writing :-
        makewindow(1,2,3,"",0,0,10,80) ,
        write ("Enter a data string" ) ,
        readln (Data) ,
        openwrite (outfile,                                % Open File For
                    "C:\\PROLOG\\data.db") ,                % Writing
        writedevise (outfile) ,                             % Change Output
                                                            % Stream
        write (Data), nl ,                                  % Write Data to
                                                            % File
        closefile (outfile) ,                               % Close File When
                                                            % Done
        writedevise (screen) ,                             % Return Output
                                                            % Stream
        write("Data successfully written.\n").
/* END */
```

This program shows the four basics needed when interacting with files. First, open a file to write to, then change the output stream to this file. Next, write the data to the file. When you have written the data, close the file, and you're done. Although it deals only with redirecting the output, redirecting the input is very similar.

There are a few things to remember when working with files. Be sure to close a file when you are finished working with it. Also, to indicate a backslash in a Prolog string, you must double the backslash (\\) in a file's path. Not giving a valid path to the file you want to interact with has caused many a programmer's headache.

Be sure to change the I/O stream back to normal when you're finished with redirection. Remember, in Prolog there's only one output stream; Prolog cannot write to two places at once. If you want to write a piece of data both to a file and to the screen, you must actually write the data out twice, once to each "write device."

File-handling Predicates

There are four different ways to open a file in Turbo Prolog. Once a SymbolicFileName is declared, it may be assigned to a file using any one of the following predicates:

```
openwrite()  
openread()  
openappend()  
openmodify()
```

Each one of these file-handling predicates works with two arguments. The first argument takes a SymbolicFileName, while the second argument points to a DOS file.

The first two file-opening predicates are straightforward; they open up a file for basic I/O. Beware: `openwrite()` will overwrite a file if the named file exists when `openwrite()` is called. If the named file does not exist when `openread()` is called at runtime, `openread()` will return an error message.

The predicate `openappend()` is fairly straightforward as well; the file is not erased when opened, but is appended instead. If the DOS file named does not exist when `openappend()` is called, an error message will indicate that the given file cannot be found.

The last file-opening predicate is meant for advanced file-handling routines. The `openmodify()` predicate is employed when writing random-access filing systems and is used in conjunction with `filepos()`. The built-in predicate

`filepos()`

points to a specific location in a file, and can therefore be used to update an existing file. The predicate `filepos()` also has an output flow pattern that allows you to see the current file position of an opened file.

Two more file-handling predicates are

`closefile()`

and

`flush()`

Each one takes a single `SymbolicFileName` for an argument. While `closefile()` simply closes the named symbolic file, `flush()` can be used to get every last bit of information out of a file buffer before it is closed.

Making a call to the predicate `closefile()` will automatically close a file when you are finished writing to it. This is in contrast to reading in a file, where you must check to see if you are at the end of the file to finish the processing. The built-in predicate

`eof()`

lets you know when you're at the end of the file (or EOF) being read. Taking a `SymbolicFileName` for an argument, `eof()` will succeed if the file pointer points to the end of the file being read. If the current file position is not at the EOF, then `eof()` will fail. Used in conjunction with `openread()`, this predicate allows you to read and process the entire contents of a file, as the following short program demonstrates.

```
/* Reading a File */  
  
Domains  
    file = infile  
  
Predicates  
    open_and_read  
    open_file_to_read  
    read_file  
    process_line ( string )
```

```

Clauses
open_and_read :-
    open_file_to_read ,
    readdevice(infile) ,
    read_file ,
    closefile(infile) ,
    readdevice(keyboard).

/* * * * * *
* Check to see if the named file exists. If it does, *
* open it to read, otherwise write an error message. *
* * * * * */

open_file_to_read :-
    existfile("C:\\PROLOG\\readme") ,
    !, openread(infile, "C:\\PROLOG\\readme").
open_file_to_read :-
    write("The named file does not exist.\n" ,
        "Press any key to terminate...") ,
    readchar(_), fail.

/* * * * * *
* Read and process the file. *
* * * * * */

read_file :-
    not( eof(infile) ) ,
    ! ,
    readln(Line) ,
    process_line(Line) ,
    read_file.                                % recursive call
read_file.

/* * * * * *
* This is where you place the file handling routine. *
* This program will read and process one line at a time. *
* * * * * */

process_line(Line) :-
    write(Line, "\n").

GOAL
makewindow(1,2,3," Read A File ",0,0,25,80) ,
open_and_read,
write("\n Press a Key..."), readchar(_).

/* END */

```

If the **README** file is in the Prolog directory of your hard disk, this program will whiz through the **README** file, dumping its contents to the screen. This doesn't do much good, though, because the file goes by too fast to read. Later, after you learn about recursive counters, you'll be able to control the flow of lines written out to the screen.

This short program contains an error-trapping routine that first tests to see if a file exists before opening it. If the file does not exist, an error message is printed and the program terminates. Error-trapping routines such as this one should be included in all your programs that deal with files.

Tail Recursion Optimization

The program just listed brings back the concept of recursion in Prolog. Notice that the user-defined predicate `read__file()` is recursively defined. Remember that a *recursive predicate* is one that is defined in terms of itself. In other words, the predicate `read__file()` makes a call to itself. Here, the recursive call is made at the end of the clause, so the call is said to be *tail recursive*. Turbo Prolog can optimize predicates that are defined as tail recursive. Internally, a tail-recursive predicate is turned into an efficient loop, making recursion a powerful tool for writing procedures that need to be repeated many times. However, simply placing the recursive call at the end of a clause does not fully define a predicate to be tail recursive.

There is no trick to defining a tail-recursive clause. Turbo Prolog can optimize recursion if no extra calls need to be processed when the recursive call is made. If Prolog can take another path, then backtracking points (bread crumbs) must be placed to indicate that another path may lead to a solution. When Prolog goes deep into a recursive cycle, these backtracking points can accumulate, creating a memory overflow problem.

For this reason, a cut has been placed in the clause defining `read__file()`. As an exercise, remove the cut from this clause and run the program. Now, if you use this program to read large files, you will run into a stack overflow. This is because backtracking points are placed on the stack each time the recursive call is made to mark a path to the untried `read__file()` clause. These backtracking points pile up on the stack until the roof of the stack is hit and the program ends with a crash.

To prevent stack overflows from occurring in this program, you must use the cut. In the `read_file()` predicate, once you have determined that you are not at the end of a file, you can cut to be sure that no backtracking points are placed on the stack that would mark the second, untried `read_file()` clause. Doing this allows `read_file()` to be a truly tail-recursive predicate (there are no untried paths when the recursive call is made), and Turbo Prolog is able to perform its optimization.

Below is another example that demonstrates the power of tail-recursion optimization. This program uses recursion to create a small loop.

```
Predicates
    run ( real )
Clauses
    run(1.0e5).                % terminating condition
    run(X) :-
        Y = X + 1 ,
        write(Y, "\n") ,
        run(Y) ,              % recursive call
        write("Done: ", Y, "! \n").
GOAL
    run(0).
```

Even though this short program doesn't do much worth noting, type the program and let it run, first with the second call to `write()`, then without this extra call.

The predicate `run()` is not considered to be tail-recursive due to the extra `write()` placed after the recursive call. Because of this, `run()` cannot be fully optimized. Backtracking points need to be placed on the stack at each recursive cycle to point out the still uncompleted subgoal. Only when the recursion reaches the terminating condition can the pointers on the stack be freed as the recursion unwinds.

In this program, each extra call to `write()` is processed after the respective call to `run()` returns. You can test this by running a modified version of the program. To do this, decrease the value in the terminating condition until the program runs without going into a memory overflow.

Although there is an inherent problem with non-tail-recursive loops, they have their place. Normal recursion can work wonders when you need to create a process that loops only a few times. The trouble crops up when a call needs to go through many cycles before a return can be made. In these cases, memory overflows can occur, and it may be necessary to code your loop in a different

fashion. A possible key to the problem may be to work out a tail-recursive solution, thus minimizing the stack needed to perform your loop. Problems that require a still larger looping process (such as a program that endlessly loops around its main body) may require that you use backtracking instead of recursion to create a process that repeats itself.

Working with Strings

A complete set of reading, writing, and string-handling predicates is supplied with Turbo Prolog. Four predicates allow the reading of different types of information, while the writing out of information is taken care of with two built-in predicates. The string-handling predicates range from the concatenation of strings to the dissection of strings.

Reading Data In

Reading whole lines of information in one fell swoop can be accomplished with the predicate

`readln()`

This predicate takes a free variable as an argument, and reads a string of characters until a carriage return is hit. A maximum of 128 characters can be read, including the carriage return, if you are reading from the keyboard. However, if the input stream has been changed from the keyboard (perhaps to a disk file or communications port), strings of up to 64K can be read and bound to a single variable.

The following three input predicates each take a free variable for an argument and will read in the types character, integer, and real.

`readchar()`
`readint()`
`readreal()`

Notice that since `readchar()` reads a single keystroke at a time, it can be used to pause in a program until the user presses a key. Because of this, it is common to see the following section of code in Prolog programs:

```
pause :- write("Press any key...") ,
        readchar(_), nl.
```

Since it does not matter which key the user hits, the anonymous variable is used as an argument to `readchar()`.

Both the predicates `readint()` and `readreal()` will fail if the user enters something other than what the predicate expects to read. The code shown below demonstrates an error-trapping routine that succeeds only after `readint()` receives an integer.

```
get_integer(Integer) :-
    write("Please enter an integer: ") ,
    readint(Integer) ,
    ! , write("The number entered is: ",Integer).

get_integer(Int) :-
    clearwindow, beep,
    write("An error was made...\n") ,
    get_integer(Int).    /* Tail Recursive Call */
```

This program will loop recursively, waiting for an integer to be entered. The code may appear clumsy at first; in fact, it is. Later, you will learn how to replace this code with a more elegant repeat/fail loop.

For now, type in the section of code and run it, after adding predicate and domain declarations, as needed. Test the program with the goal

```
get_integer(Integer)
```

As the program prompts for input, type in various characters and strings before actually entering an integer. Notice that when at last you do type an integer, the program succeeds and outputs your result to the screen. To see how this program works, place the program in Trace mode and watch Prolog process the goal.

Each time a recursive call to `get_integer()` is made, that call must be satisfied before the very first call to `get_integer()` can succeed and return a value. When at last you enter an integer, the pro-

gram will "unwind" from the recursion, returning the integer value that was input back to the original call.

As an exercise in tracing, output the results of the trace to the printer, and match each CALL made to `get_integer()` with its respective RETURN. To get a printout, bring up the Printer Status window (press ALT-P) and turn the printer status on.

If you wish to get a quick dump of the trace, bring up the Trace Status window (press ALT-T), and turn the Editor window off. Now when the program is run Turbo Prolog will output the trace without waiting for you to hit the F10 key at each call. If the printer is turned on, the entire trace as shown in Figure 10-1 will be output to the printer.

Writing Data Out

The ability for a programming language to read in data is just not enough. For a language to be really useful, one must be able to output the results gained during the processing of a program.

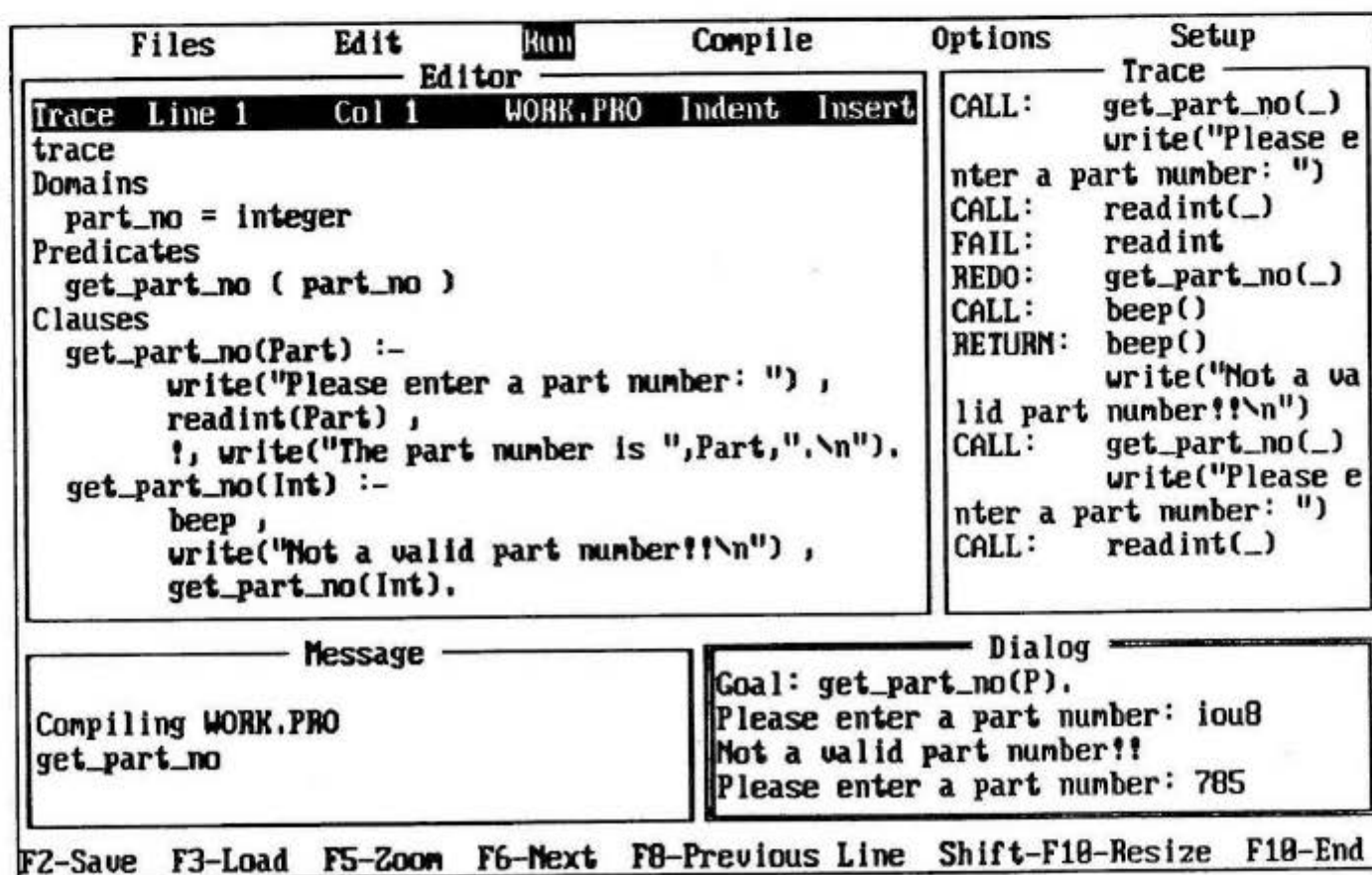


Figure 10-1. Tracing a recursive call

Turbo Prolog offers two built-in predicates that handle the writing out of data (the first of which you are already familiar with):

`write()`

and

`writeln()`

Depending on where the current output stream is pointing, calling one of these predicates will result in data being written out to a particular location. By default, the output stream points to the computer screen, writing data into the currently active window. Switching the destination of outgoing data is done with `writeln_device()`; thus, you can change the direction of the output to a disk file, the printer, or even one of your computer's I/O ports.

The predicate `write()` will take any number of arguments; however, each one must be separated by a comma. Each argument given to `write()` must be a valid Prolog term. Even complex data objects are allowed, although they must be bound to a variable before being written out. Also, when the `write()` predicate is evoked, the argument list must not contain any free variables or else your program will end with a compiler error. The following example demonstrates the power of the `write()` predicate:

```
Domains
    foo = f(integer)
    integerlist = integer*

Predicates
    goo ( foo )

Clauses
    goo( f(100) ).

GOAL
    makewindow(3,6,5,"",0,0,12,60) ,
    goo(X) ,
    List = [1,2,3] ,
    write("The value of X is: ",
        X, "\nNotice that it is not:\n\n\t",
        1.5, "\n\t", foo, "\n\t", '\145',
        " or even\n\tfoo(", List, ")!\n").
```

This example shows the writing of symbols, reals, strings, characters, lists, and compound terms. String arguments can be used

extensively inside of calls to `write()`, thus allowing for flexible output.

The example also shows that the arguments to `write()` can be split up onto several lines of actual Prolog code. Splitting the arguments in this way has no effect on the line spacing when the `write()` is executed, but it allows your program code to be easily read. Actually, you can split up any predicate's argument list in this manner, starting a new line after the comma that separates any of the predicate's arguments.

Notice as well the inclusion of the special escape character symbolized by the single backslash (`\`). If you type this program in and run it, you will see that the output is indeed spaced out over several lines. This is due to the special formatting codes available for writing strings.

Turbo Prolog allows several control codes to be included in strings that can be written out. Each one must be prefaced with the backslash to indicate that a special control character follows. Because of this, a single backslash never represents an actual backslash. In Table 10-2 you will see several codes that Turbo Prolog recognizes as special control characters, along with their associated print values.

When a backslash precedes a character that is not a control character, the character after the backslash prints out, unaffected. In this way it is possible to print several characters that would otherwise present a printing problem. The backslash itself is one such character. The double quote mark is another, which, when presented without a preceding backslash, indicates that the end of the string has been reached.

Escape Character	Print Value
<code>\n</code>	New line
<code>\t</code>	Tab
<code>\b</code>	Backspace
<code>\nnnn</code>	ASCII character representation
<code>\"</code>	"
<code>\'</code>	'(in characters only)

Table 10-2. *Special Formatting Characters*

The second output predicate in Turbo Prolog is `writeln()`, short for “write formatted.” If you are familiar with the C programming language, you will learn how to use `writeln()` quickly because it is analogous to the `printf()` function in that language. Like `write()`, `writeln()` can take any number of arguments. However, the difference is that `writeln()` must have a string as its first argument. This first argument is known as the *format string* and contains various *format specifiers* that allow Prolog to print out values in a formatted manner.

To indicate that a value is to be printed in a specific format, place a percent sign (%) inside the format string where you want the value to appear. Following the percent sign, place the format code that tells Prolog how to handle the value. The formatting codes available with the `writeln()` predicate are shown in Table 10-3 and take the general form of % — m.pf. Print values are supplied by the arguments located after the opening format string.

Each value argument must have a corresponding format specifier located in the string to be printed. The first specifier in the string takes the first value argument, printing it out in the matching format, and the second specifier matches the second value argument, printing it out as well. If you don't supply an equal number of values to the specifiers located in the string, you can expect your output either to look funny or to show an error during compilation. To show how `writeln()` works, compile an empty editor to memory (this is a handy technique used to test how built-in predicates work). At the Goal: prompt, type the following compound goal:

```
W = good_day ,
X = 1.23e-3 ,
Y = 3.141,
Z = 129 ,
writeln("Write a symbol: %, \nAnd now numbers: \n%f \n%8.2 \n %c",
W,X,Y,Z).
```

String Handling

This section covers various string-handling predicates that Turbo Prolog offers. To start off, the predicate

`str_len()`

takes two arguments. The first argument to `str_len()` must be bound to a string. If the second argument is free, the length of the string will be returned bound to this argument. If the second argument is bound to an integer, then `str_len()` will succeed if the integer is equal to the length of the input string.

Format Specifier	Description
%	Position in string where value is to be placed
-	Indicates that the value is to be left justified in the field (default is right justified)
m	Decimal number specifying the minimum field length
.p	Decimal number that specifies the maximum number of characters to be printed from a string or that specifies the precision of a floating point number
f	One of the value specifiers shown in the table below
Value Specifier	Description
f	Real numbers in fixed decimal notation
e	Real numbers in exponential notation
g	Real numbers in the shortest printable notation (of either e or f)
d	Characters or integers as a decimal number
u	Characters or integers as an unsigned integer
x	Characters or integers as a hexadecimal number
c	Characters or integers as a character
s	String or symbol
R	Database reference number
X	Strings or a database reference number as a long hexadecimal number

Table 10-3. *Specification Format %-m.pf*

The predicate

`concat()`

is a very powerful and useful string-handling predicate. Used both to stick strings together and to take them apart, the predicate can handle an assortment of flow patterns. Depending on the arguments that are free and that are bound when called, `concat()` will either take apart the string entered or it will put two strings together. For example, the following section of code binds the variable "X" to the string value "Today":

```
concat("To", "day", X)
```

The example below shows how `concat()` can be used to take a string apart:

```
concat(STR, " today", "Good day today")
```

Here, the variable "STR" is bound to the value "Good day".

While `concat()` can be used to dissect strings, three predicates are specifically designed for this task:

```
frontchar()  
frontstr()  
fronttoken()
```

These predicates are mainly used for parsing strings in advanced programming applications. Since string parsing is beyond the scope of this introductory book, refer to the *Turbo Prolog Owner's Manual* for a more complete discussion of dissecting strings.

The predicate

`isname()`

takes a single string argument, and succeeds if this argument is a valid name according to Turbo Prolog syntax.

Another useful predicate to know about is

`display()`

Taking a single string value, this predicate displays the string in the currently active window. Much like the `edit()` predicates introduced in the next section, `display()` can be used to show text files up to 64K in length. Note that the files are in a Display only mode; no editing of the text is allowed. However, a full range of cursor control keys is available, allowing for the text to be scrolled or paged up and down. Searching for strings is also allowed, making `display()` a versatile way to view text.

The last string-handling predicate to be introduced is

`format()`

This predicate behaves exactly like `writeln()`, except that `format()` returns the string in a variable instead of writing out the string to the current output stream. Because of this, the first argument supplied to `format()` must be a free variable when called, and the format string is moved over to the second argument (unlike `writeln()`, where the first argument is the format string). The print values, as in `writeln()`, follow up the format string in the list of arguments. For more details on how to format strings with `format()`, refer to the section concerning the `writeln()` predicate.

Pulling in the Turbo Editor

Turbo Prolog allows you to create programs with strong user interfaces by giving you a set of built-in editing commands. With the ease of making a single call, you can have the power of the Turbo editor included in your programs.

Version 1.1 of Turbo Prolog provided two built-in editing predicates that allowed the creation or editing of ASCII text files. They were

`edit()`

and

`editmsg()`

The simplest form of the editor can be called with the two arity

versions of `edit()`. With the use of the `file__str()` predicate, a “quick-and-dirty” editor can be made with just two short calls:

```
make__editor :-
    edit("",Text) ,
    file__str("C:\\PROLOG\\test.dat", Text).
```

This short program allows you to edit an empty string, placing the text entered into the DOS file TEST.DAT.

The predicate

`file__str()`

has two flow patterns, allowing it to be used in two different ways. The first flow pattern (i,i), is shown in action in the `make__editor()` clause. Since both arguments have input flow values, both arguments must hold a value when the predicate is called. The string in the second argument is written to the DOS file, which is indicated by the first argument.

The other version of `file__str()` uses an (i,o) flow pattern. Used in this fashion, the first argument must be bound to a string (a valid DOS file name), while the second argument must be a free variable. When `file__str()` is used with this flow pattern, it gives you quick access to ASCII text files up to 64K in length. The predicate allows you to read an entire text file into a single string variable with the quickness of making a single call. The usefulness of this predicate is shown in a program presented at the end of this chapter, which creates and updates a to do list on your computer system.

In version 2.0 of Turbo Prolog, the two editing predicates mentioned above have been combined into a single editing command. While both of the previous editing predicates still exist, they are left in mostly for backward compatibility. Unless you plan to use a minimal editor in your program, you should use the 13-arity version of `edit()` for your editing applications. The arguments to the long version of `edit()` take the following values:

```
edit( InPut,
      OutPut,
      Header__1,
      Header__2,
```



```
Window__msg,  
Beginning__pos,  
Help__file,  
Display__mode,  
Indent,  
Overwrite,  
Text__mode,  
Return__pos,  
Status )
```

As you can see, a fairly flexible and powerful editor is provided with Turbo Prolog. Whenever the editor is called, the currently active window becomes the spot where the editing takes place, as the input string is placed into this window. The string that is passed as the first argument to `edit()` becomes the input string, while the second argument returns the string after it has been edited. The input string may be up to 64K in length, or it may be bound to an empty string to represent a new file to be created.

The third, fourth, and fifth arguments present a message in the editor, each at a different window location. The `Header__1` and `Header__2` arguments each present a message in the header of the Editor window while the editor is being used. The `Window__msg` argument takes a string, which is presented at the bottom of the Editing window and remains there until a key is pressed.

`Beginning__pos` places the cursor at a specified position in the file when editing begins. It is used in conjunction with the `Return__pos` argument, which returns the position of the cursor when the editor is exited. These arguments can be used if you wish to save the cursor position in a file you're working on.

If you want to make a help file available, the seventh argument allows a specific help file to be presented when the F1 key is pressed.

The eighth, ninth, tenth, and eleventh arguments of `edit()` each have a set mode in the editor for when they are brought up. Each argument takes either a zero (0) or a 1 to set its respective mode. The "Display mode" argument allows you to specify whether any editing of text can take place. When called with a zero in this argument, the editor is set for display only, while 1 allows the text to be edited. "Indent" sets the Auto-indent mode to on or off (1 is on), and "Overwrite" sets the Overwrite/Insert mode (zero sets Overwrite mode to on). A powerful feature of the `editor()` allows

you to place it into Word-wrap mode. A 1 in Text__mode allows the text in the Edit window to word wrap when it hits the right side of the window frame. Note that you may toggle all of these modes once the editor is brought up.

Last, the thirteenth argument, "Status", indicates how the user exited the editor. If the user hit an F10 to leave the editor, a zero is returned. A 1 is returned if the ESC key was hit to exit. This feature allows the user to abort any edit that had been made. The program shown at the end of this chapter shows how the user may abort the edit that has taken place.

A Few More File-handling Predicates

To finish this chapter's long list of new predicates, here are some other useful file-handling predicates that can be put to good use. First, the predicates

`existfile()`

and

`deletefile()`

both take a single string argument that must be bound to a valid DOS file name. The predicate `existfile()` will fail if the DOS file given is not found, but it will succeed if the file named is found to exist. The `deletefile()` predicate will succeed as long as the string passed is a valid DOS file name. Both predicates will return errors if you attempt to pass DOS wildcard characters as part of the file name.

Last, the predicate

`dir()`

has a short version and a long version. The short `dir()` predicate is called with three arguments, and allows the user to select a DOS file from the path and file specification (file-spec) given. The first two arguments in the call must be bound to string values; together

they indicate a disk path and file-spec for the directory listing you wish to give. The first argument selects the drive and path, and must represent a valid path found on your computer system. The second argument selects the file-spec you wish the directory listing to show. Any valid DOS file-spec is allowed, including the wildcard characters “?” and “*”. For more information on DOS file-specs, refer to your DOS manual.

When `dir()` is called, a directory listing of the path and file-spec given is displayed in the currently active window. You may then use the cursor control keys to move and select a file from the list (this is much like the directory listing you get when you ask to load a file from the Turbo Prolog environment). The third and last argument to `dir()` must be free when the predicate is called, and returns the file name selected from the given directory.

The longer version of `dir()`, a six-arity predicate, is the same as the three-arity command, except that the three extra arguments allow various switches that modify the directory listing given to be set. The switch is set by placing either a zero (0) or a 1 in the arguments position. The fourth argument specifies whether sub-directories should be shown in the directory listing. A zero in this position indicates that the user can “walk” through the tree directories in search of a file to choose from. The fifth argument allows the user to set the file mask of the directory given (by pressing the F4 key), while the sixth argument shows the current path in the current window (if it is set to zero).

Seeing It All Work Together

The predicates presented in this chapter can be put together into a program that provides for a full file-handling system. Below is a small utility that can be placed in your `AUTOEXEC.BAT` file. Simply type in the code and compile the program to an `.EXE` file. If you add the program to your `AUTOEXEC.BAT` file, a to do list will be pulled up each time you start your computer. The program allows you to add and delete items from the list as much as you like. The program also provides an escape route, so that you can abort any editing you may have inadvertently done to your list.

```

/* To Do List */
Domains
    file = to_do

Database
    position ( integer )

Predicates
    edit_file ( string, string, integer )
    get_date ( string )
    get_file ( string )
    get_position ( integer )
    save_file ( string, integer )
    save_position ( integer )
    run

Clauses
    get_file(To_do) :-
        existfile("C:\\PROLOG\\to_do.lst") ,
        ! ,
        file_str("C:\\PROLOG\\to_do.lst", To_do) .
    get_file("To Do List:\\n\\n").

    edit_file(Old, New, Ret_code) :-
        get_position (Pos) ,
        get_date(Date) ,
        edit(Old, New, "", Date,
            "Press F10 to Save; ESC to Abort Edit",
            Pos, "C:\\PROLOG\\prolog.hlp" ,
            1, 1, 1, 0, New_pos, Ret_code) ,
        clearwindow ,
        save_position(New_pos).

    get_position(Pos) :-
        existfile("C:\\PROLOG\\position.dat") ,
        ! , consult("C:\\PROLOG\\position.dat") ,
        position(Pos) .
    get_position(13).

    get_date(Date) :-
        date(Y,M,D), str_int(Y1,Y) ,
        str_int(M1,M), str_int(D1,D) ,
        concat(M1,"/",M2), concat(M2,D1,D2) ,
        concat(D2,"/",D3), concat(D3,Y1,Date) .

    save_position(Pos) :-
        retractall( position(_) ) ,
        asserta( position(Pos) ) ,
        save("C:\\PROLOG\\position.dat").

    save_file(To_do, Code) :-
        Code = 0, ! , /* save if F10 was used to exit */
        file_str("C:\\PROLOG\\to_do.lst", To_do) ,
        write("File Saved.\\n\\n").
    save_file(_,_) . /* ESC was pressed, don't save. */

Goal
    makewindow(1,2,3," To Do List ",0,0,24,80) ,
    get_file(To_do) ,
    edit_file(To_do, New_to_do, Ret_code) ,
    save_file(New_to_do, Ret_code) ,
    write("Press any Key...") ,
    readchar(_).

/* END */

```

You should understand most of the processing that goes on in this program, although a few sections may seem unfamiliar. The most obvious newcomer is the inclusion of a database declaration section placed after the Domains section. Although the database is covered in detail in the next chapter, it may help to see the database in action now. The database can be used to store values obtained during the execution of a program. In the program above, the database is used to save the editor's file position from one run of the program to the next.

The other parts of the program should be fairly easy to follow. When run, the subgoals under the Goal section become the primary goals for the program to solve. The first thing the program does is to make a window. After this, processing continues with a call to the user-defined predicate `get_file()`.

The predicate `get_file()` returns a `to_do` string to edit. If the `TO_DO.LST` file already exists, then it is returned in the variable `To_do`. If the file has not yet been created, then the string "To Do List:\n\n" is returned as the string to edit. Notice how the predicate `file_str()` is used to read in the `TO_DO.LST` file, if the file is found to exist.

After a string is returned from `get_file()`, the editor is then called into action. Note how the date, returned from the system clock, is put into presentable form through multiple concatenations. After it is formatted, the date is placed in the header of the Edit window.

Take a look at the user-defined predicates `get_position()` and `save_position()`. Both use the database in their processing.

The last section in the program is a predicate that tests the return status of the editor. If the user exited using F10, a new to do list is saved to disk. If ESC was used for exiting, the edited version of the list is thrown away.

As an exercise in following program flow, place this program in the Trace mode and run it two or three times. This will help you to see how the flow through the program changes as the values of the arguments change.

11 Knowledge Bases in Turbo Prolog

In Turbo Prolog, the database takes on many different identities. Collectively, the facts and rules that you write in the Clauses section of a program are sometimes referred to as the program's database. In Prolog, there is no distinction between the program and the data that the program operates on.

In a conventional sense, a database is a set of facts that a program uses to come up with solutions to a problem. During the processing of a program, a call can match the head of a rule and, in turn, the subgoals in that rule can match with other rules. In the end, however, a fact must be encountered before any query can return from a call. Seen in this light, a database is a set of facts that enables Prolog to come up with positive solutions to the queries made to the program.

Data Representation

Conventional programming languages create a chasm between the program and the data the program feeds upon. To deepen this gap, most programs keep their databases in separate files from the pro-

gram itself. Data is read into the program and output is generated as a result of processing.

This scenario is not quite the same in a Prolog program, because the concepts that Prolog operates with are a bit different. In Prolog, the database a program works with is often known as a *knowledge base*. Facts define the knowledge that a program relies on as being true. While certain facts are written to enhance the rules you define (such as the terminating condition of a recursive clause), other facts are written in groups and serve the same purpose as a database in conventional programming language.

Static Data

In Prolog, a set of facts describing knowledge can be *static*, meaning that the data will not change from one run of the program to the next. For instance, you might wish to define a set of facts that lay out the continents of the world:

```
continent(asia).  
continent(north__america).  
continent(south__america).  
continent(australia).  
continent(africa).  
continent(antarctica).  
continent(europe).
```

This set of data is static because it is known that the data will not change. A set of data like this can be *hard coded* into the program, meaning that the actual program code contains these facts. Only a programmer capable of modifying actual program code is able to change knowledge that has been hard coded into a program.

Dynamic Databases

In Turbo Prolog, there are several kinds of databases. Static data-

bases can be written to define hard-and-fast knowledge. Turbo Prolog also supports both *internal* and *external* databases, and each type has several varieties. While external databases can be used to mimic the databases in procedural programs, internal databases define what is known as a *dynamic*, or changing, database.

The difference between static and dynamic databases is that dynamic databases have the ability to be added to or deleted from as the need arises. For example, when writing a program to keep the addresses of your friends, you wouldn't hard code the data, since the program would be unreliable as soon as your friends changed their addresses. Instead, you would use a dynamic database because the data can be changed as the conditions surrounding the program change.

Internal Databases

One trademark of a Prolog program is its ability to modify itself during the course of operation. Adding and deleting facts from the database while the program is running changes the solutions the program comes up with. This flexibility is the basis of many artificial intelligence (AI) applications, because it allows a program to "learn" by adding new facts as they are proven true and removing old ones when they have become outdated.

A dynamic database can be easily incorporated into your programs with the use of an internal database. Knowledge stored in an internal database consists of a set of facts that is either *asserted* or *retracted* during the operation of a program. Internal databases can also be saved to a file or *consulted* into a program when needed. The facts in the internal database are kept in sequential order, much like the other clauses written in your program.

Internal Database Declarations

Facts that are used in an internal database are declared in the Database section of the program. This section (or sections) lies between the Domains section and the Predicates section, and the declarations made there are much like those made in the Predicates section. Special user-defined domains can be used, and the Database section can hold as many different database declarations as your program needs. A set of database declarations looks something like this:

Database

```
database__fact__1 ( integer )  
database__fact__2 ( symbol )
```

Asserting and Accessing Database Facts

Turbo Prolog supports several built-in predicates to manipulate an internal database. First, to get a fact into the database, a predicate from the *assert* group is used:

asserta()

assertz()

assert()

Since the facts are stored sequentially, you have the choice of placing a new fact at either the top or bottom of the existing database. The first predicate, *asserta()*, places a fact at the top of the database stack. The other two predicates, *assertz()* and *assert()*, are analogous: Each asserts the respective fact at the bottom of the database.

The following program demonstrates how database facts are declared and placed into an internal database. It also shows how the facts in the database are accessed to get information out:

Domains

```
person = symbol  
parents = person*
```

Database

```

father ( person, person )
mother ( person, person )
female ( person )
male ( person )

```

Predicates

```

parent ( person, person )
get_fathers
get_mothers
fathers_children
mothers_children

assert_father ( person )
assert_fathers_children ( person )
assert_mother ( person )
assert_mothers_children ( person )
assert_sex ( person, char )

get_sex ( char )
check_sex ( char )
write_children ( person )

list_parents ( parents )
remove_dups ( parents, parents )
member ( person, parents )

```

Clauses

```

/* * * * * *
 * Family Relationships *
 * * * * * */

parent(Parent, Child) :-
    father(Parent, Child).
parent(Parent, Child) :-
    mother(Parent, Child).

/* * * * * *
 * Enter Fathers and their Children *
 * * * * * */

get_fathers :-
    write("Please enter a fathers name.\n (RETURN to quit): "),
    readln(Father) ,
    Father <> "" , ! , nl , % fail if no new father
    assert_father(Father) ,
    assert_fathers_children(Father) ,
    clearwindow ,
    get_fathers.
get_fathers.

assert_father(Father) :-
    male(Father) , ! . % succeed if father is in database
assert_father(Father) :-
    assert( male(Father) ) . % otherwise, assert him in

assert_fathers_children(Father) :-
    writef(" Enter a child of %.\n (RETURN to quit): ",Father),
    readln(Child) ,

```



```

    Child <> "", !, % fail if no new child
    assert( father(Father, Child) ) ,
    assert_fathers_children(Father).
assert_fathers_children(_).

/* * * * * *
* Enter Mothers and their Children *
* * * * * */

get_mothers :-
    write("Please enter a mothers name.\n (RETURN to quit): "),
    readln(Mother) ,
    Mother <> "", !, nl ,
    assert_mother(Mother) ,
    assert_mothers_children(Mother) ,
    clearwindow ,
    get_mothers.
get_mothers.

assert_mother(Mother) :-
    female(Mother), ! ; % succeed if mother is in database
    assert( female(Mother) ). % otherwise, assert her in
assert_mothers_children(Mother) :-
    writef(" Enter a child of %. \n (RETURN to quit): ", Moth
    readln(Child) ,
    Child <> "", ! ,
    assert( mother(Mother, Child) ' ) ,
    assert_mothers_children(Mother).
assert_mothers_children(_).

/* * * * * *
* Get the Sex of Children who are not Parents Themselves *
* * * * * */

mothers_children :-
    mother(_, Child) , % get a child
    not( male(Child) ) , % check if child's sex is known.
    not( female(Child) ) ,
    writef("Is % a male or female (M/F)? ", Child) ,
    get_sex(Sex) , % get child's sex
    assert_sex(Child, Sex) , % assert child's sex
    fail ; true.

fathers_children :-
    father(_, Child) ,
    not( male(Child) ) ,
    not( female(Child) ) ,
    writef("Is % a male or female (M/F)? ", Child) ,
    get_sex(Sex) ,
    assert_sex(Child, Sex) ,
    fail ; true.

get_sex(Sex) :-
    readchar(Sex1) ,
    upper_lower(Sex1, Sex) ,
    check_sex(Sex) ,
    write(Sex1, "\n"), !.
get_sex(S) :- get_sex(S).

check_sex('m') :- !.
check_sex('f') :- !.
check_sex(_) :- beep, fail.

assert_sex(Child, Sex) :-

```

```

    Sex = 'f' ,
    assert( female(Child) ), ! ;
    assert( male(Child) ).

/* * * * * *
 *   Print Out Parents and Children *
 * * * * * */

list_parents([]) :- !.
list_parents([Parent|T]) :-
    writef("The following are the children of %:\n", Parent) ,
    write_children(Parent) ,
    list_parents(T).

write_children(Parent) :-
    parent(Parent, Child) ,
    writef("\t%\n", Child) ,
    fail ; nl.

/* * * * * *
 *   Utility Predicates *
 * * * * * */

remove_dups([],[]) :- !.
remove_dups([H|T], T1) :-
    member(H,T), ! ,
    remove_dups(T,T1).
remove_dups([H|T],[H|T1]) :-
    remove_dups(T,T1).

member(X,[X|_]) .
member(X,[_|T]) :- member(X,T).

GOAL
    makewindow(1,2,3," Family Members ",0,0,25,80) ,

    % Get fathers and mothers and their children
    makewindow(2,6,5," Fathers ",1,1,23,39) ,
    get_fathers ,
    makewindow(3,3,2," Mothers ",1,40,23,39) ,
    get_mothers ,
    removewindow ,
    removewindow ,

    % Get the sex of children who are not parents
    makewindow(4,3,2,"",2,1,21,78) ,
    mothers_children ,
    fathers_children ,
    removewindow ,

    % List parents and their children
    findall(Parent, parent(Parent,_), Parents) ,
    remove_dups(Parents, Parents_1) ,
    list_parents(Parents_1).

```

Don't let the size of the program fool you into thinking that it's too complex for you to figure out. The program prompts for and stores a knowledge base of parents and keeps track of their children. The program also keeps track of who is male and who is female.

Type the program in and run it once or twice to get a feeling for what the program does. Then take a look at the code, and try to piece together how the processing takes place. To unravel the program, start with the Goal section, and follow the calls and returns made there. Trace the sections that give you trouble to gain an understanding of the way things work.

The logic of certain processes shouldn't be too difficult to figure out. For example, it is easy to find the sex of a father or mother, but a child's sex is unknown unless the child in question is also a parent. Because of this, the user of the program will be prompted for the sex of any children who are not also parents. This processing can be seen in the clauses defining the predicates `mothers__children()` and `fathers__children()`.

Asserting and accessing facts in the database can be seen in the procedure defining `assert__father()`. This set of clauses first checks to see if the knowledge base already contains the father named. Since all fathers are male, this is checked by calling `male(Father)`, with "Father" bound to the current father being processed. If the query doesn't succeed, then the father is added into the knowledge base with the call

```
assert( male(Father) ).
```

Notice that querying an internal database fact is no different than checking to see if that fact is included in the Clauses section of the program. Prolog makes no distinction between facts asserted into an internal database and facts written directly into the body of the program.

Asserting a fact into the database is shown in the second clause of `assert__father()`. Predicates that manipulate database facts are unique in that they take an actual Prolog fact (with parentheses, arguments, and all) as an argument to the predicate. The one restriction regarding facts and an internal database is that all arguments must be instantiated to values before they can be asserted into the database; no free variables are allowed in the database.

A new logical construct, which uses the OR statement, is introduced in this program. The OR statement (;) is used to define an alternate set of subgoals when a first set of subgoals fails. Compare the two procedures defining `assert__father()` and `assert__mother()`:

```

assert_father(Father) :-
    male(Father), !.
assert_father(Father) :-
    assert( male(Father) ).

assert_mother(Mother) :-
    female(Mother), ! ;
    assert( female(Mother) ).

```

Aside from the fact that one predicate operates with fathers and the other with mothers, the two procedures are identical. Since the two clauses defining `assert_father()` make use of the same rule head, the second clause is actually an alternate solution to the first. The cut is important in this definition, because without it `assert_father()` would be nondeterministic. If something in the program failed, backtracking could end up asserting a duplicate fact into the database.

A more straightforward example of asserting predicates follows. It shows how facts asserted with `asserta()` are placed sequentially before any of the previous facts in the database. The predicates `assert()` and `assertz()` are used to place facts at the end of the knowledge base:

```

Database
    number ( integer )

Predicates
    run

Clauses
    run :-
        assert( number(1) ) ,
        assertz( number(2) ) ,
        asserta( number(3) ) ,
        fail.
    run :-
        number(X) ,
        write(X), nl ,
        fail ; true.

GOAL
    run.

```

This program uses backtracking to step through each fact in the database. Again, the **OR** is used to provide an alternate solution to a call. In the second `run()` clause, the **OR** ensures that the clause will succeed by placing the first set of subgoals in disjunction with the built-in predicate `true()`. This makes sure that, no matter what, a call to `run()` will succeed.

It is okay to use disjunction in this way. However, care should be taken not to use this form of logic with clauses that return values. When this is done, you will receive an error message stating that a variable is not bound on its return from a call.

Saving and Consulting the Database

In most programming applications you will want to save the data you have added to the internal database. The predicate

`save()`

does just this. Taking a single string argument, `save()` will write the facts in the database to the DOS file indicated in the string. If the file named in the string already exists, the file will be overwritten when the database is saved.

It is useful to read data into programs that you plan to work with. Almost identical to `save()`, the predicate

`consult()`

identifies a DOS file name and “reads” the file into the internal database. Consulting a database can be very tricky. Care must be taken to ensure that the file consulted has perfect syntax; otherwise, an embarrassing runtime error will occur. To guarantee that your database will consult correctly, use the program shell shown below. Creating a file of database facts with a program like this will warrant that the file can be successfully consulted into a program.

```
Domains
  [ Declare User Defined Domains ]

Database
  [ Declare Database Facts ]

Clauses
  [ Define Database Facts as Clauses ]

GOAL
  save("Database.dba").
```


This program shell makes use of a feature in Turbo Prolog that automatically asserts any database predicates that are hard coded into the program. The program begins by asserting all the facts into the internal database. Next, when the Goal section is encountered, all the database facts are saved to the file DATAFILE.DBA. The program works with the idea that any file written with the `save()` predicate is assured of having the proper syntax to be consulted by a program.

With the use of the predicates `save()` and `consult()` programs can be created that retain information gained during program runs. A program can be given the aura of intelligence by creating processes that use information gained from previous inferences. A program such as this is essentially "learning" from the facts it has previously proven to be true, which gives new meaning to the term artificial intelligence.

Retract and Retractall

A dynamic database wouldn't be truly dynamic if you couldn't delete the facts that no longer proved to be useful. Turbo Prolog supplies two built-in predicates that allow you to purge facts from the internal database:

`retract()`

and

`retractall()`

While the predicate `retractall()` allows you to delete a set of facts in one fell swoop, `retract()` is more selective with the facts it deletes. One major difference between the two predicates (and one that you should be especially watchful for when programming), is that `retractall()` is deterministic while `retract()` is nondeterministic. This means that `retract()` generates backtracking points when it is called. Sometimes this can prove to be useful, but at other times you'll wonder how your program came up with such an incorrect result.

Using Retract to Return Values

A property of `retract()` allows it to return values from the facts it is deleting. The following program shows how this feature can be put to use:

```
Domains
    item = symbol
    quantity = integer

Database
    part(item, quantity)

Predicates
    reorder_parts

Clauses

/* * * * * *
 * Database of parts *
 * * * * * */

part("js-394", 15).
part("js-445", 2).
part("js-278", 0).
part("ks-144", 4).
part("ks-166", 0).
part("ks-715", 12).

/* * * * * *
 * Process Database *
 * * * * * */

reorder_parts :-
    retract( part(Part_no, 0) ) ,
    writef("Part number % is out of stock!\n", Part_no) ,
    fail.
reorder_parts :-
    nl ,
    part(Part_no, Q) ,
    Q <= 5 ,
    writef("Part number % needs re-ordering.\n", Part_no) ,
    fail.
reorder_parts :-
    write("*** End of Report ***\n").

GOAL
    makewindow(1,2,3," Reorder Parts ",0,0,25,80) ,
    reorder_parts.
```

This program first deletes all the parts that have no inventory. A report is then printed based on the facts that have been retracted. Notice how the nondeterministic nature of `retract()` generates multiple solutions for facts that have zero stock on hand. Next, parts with a stock of five or less are flagged for reordering, and another report is printed.

Using `retractall()`, all the parts with a deleted inventory could have been retracted with the single call

```
retractall( part(—,0) )
```

By using this call, though, the individual part numbers could not have been returned and used to generate a report.

Saving Specific Database Facts

Sometimes it may be necessary to save a certain group of facts separate from a larger group of database facts. For instance, suppose you have a database that has several fact groups, but you only want to save one particular group. The following program shows how a selective group of facts can be picked out of a database and saved to a file. The saved file can then be consulted in another process.

Domains

```
file = datafile
name = n(string, string)
item = symbol
status, quantity = integer
```

Database

```
customer(name, status)
part(item, quantity)
```

Predicates

```
output_overdue_customers
write_overdue_customers
```

Clauses

```
/* * * * * *
 * Database of Customers and Parts *
 * * * * * */

customer( n("Tim", "Wheeler"), 1).
customer( n("Fred", "Schlicting"), 0).
customer( n("Peter", "Miller"), 1).
customer( n("Matt", "Fasburg"), 1).
customer( n("Victor", "Fresco"), 0).
customer( n("Laurel", "Doran"), 0).

part("js-394", 15).
part("js-445", 2).
part("js-278", 0).
part("ks-144", 4).
part("ks-166", 0).
part("ks-715", 12).
```

In this program, even though there are many facts in the database, only the customers with a status of 1 are output to disk.

Multiple Internal Databases

If you have a single database, and you want to manipulate one set of facts in that database, then you must construct something similar to the `output_overdue_customers()` predicate shown in the previous program. With the use of multiple databases, an entire group of facts can be retracted or modified without worrying about the other facts that are in the internal database.

Manipulating Multiple Databases

Each built-in predicate that handles the internal database has both a single- and a double-arity version. The single-arity version of these predicates addresses facts that haven't been given a specific database name. The two-arity version of these predicates has room for a database name, allowing for specific groups of data to be accessed. The group of internal database predicates is

`asserta()`

`assertz()`

`assert()`

`retract()`

`retractall()`

`save()`

`consult()`

While the single-arity version of these predicates uses the default database, the double-arity version operates on the database specified in the second argument.

Declaring Your Own Databases

Creating your own database name is easy. A special name is given to the database in the declaration of the Database section. Below, a special database is declared, with the name "parts", while another has a database domain of "prices:"

Domains

```
item = symbol
quantity = integer
amount = real
price = p(item, amount)
```

Database - parts

```
part(item, quantity)
```

Database - prices

```
cost(price)
```



```

Predicates
consult_prices
consult_parts
add_new_parts
look_up_prices
save_new_data

Clauses
consult_prices :-
    existfile("PRICES.DBA"), !, % if file exists,
    consult("PRICES.DBA", prices). % consult it
consult_prices. % otherwise, do nothing

consult_parts :-
    existfile("PARTS.DBA"), !,
    consult("PARTS.DBA", parts).
consult_parts.

✓ add_new_parts :-
    write("Enter a part number: ") ,
    readln(Item) ,
    Item <> "", ! ,
    write("Enter quantity on hand: ") ,
    readint(Quantity) ,
    write("Enter the price per item: $") ,
    readreal(Cost), nl ,
    assert(part(Item, Quantity), parts) , % assert part
    assert(cost(p(Item, Cost)), prices) , % assert price
    add_new_parts.
add_new_parts :-
    write("Press a key to continue...") ,
    readchar(_) ,
    clearwindow.

look_up_prices :-
write("Item Number      Quantity      Price Per      Total Cost\n",
    "=====      =====      =====      =====\n"),
    fail.
look_up_prices :-
    part(Item, Quantity) ,
    cost(p(Item, Cost) ) ,
    Total = Cost * Quantity ,
    writef("%11s %12d %13.2f %14.2f\n" ,
        Item, Quantity, Cost, Total) ,
    fail.
look_up_prices.

save_new_data :-
    save("PRICES.DBA", prices) , % save prices database
    save("PARTS.DBA", parts). % save parts database

GOAL
makewindow(1,2,3," Parts & Cost ",0,0,25,80) ,
consult_prices ,
consult_parts ,
add_new_parts ,
look_up_prices ,
save_new_data.

```

Now the facts describing the predicate `part()` all belong to the database named "parts". Any time you want to manipulate a `part()` with one of the built-in database predicates, you must use a two-

arity version of that predicate. However, calling the `part()` predicate is no different than before; simply make a query to the predicate in the database you wish to access.

Database Domains

Each internal database has a specific domain associated with it. A *domain* is the name you have given to the database in the actual declaration of the Database section. In the above example, the facts describing the predicate `part()` are all from the domain "parts." By default, the database domain that Turbo Prolog uses is "dbasedom." If you don't give a name to your Database section, Turbo Prolog will automatically create the following declaration:

Database - dbasedom

<... database declarations ...>

In your program you can have as many internal Database sections as you wish. The only restriction is that no two databases can be declared to have the same name; in other words, no two databases can be declared that are described by the same domain.

Using the Database to Store Values

The internal database can be effectively used to store values that you have come up with during the processing of a program. For instance, recall the looping structure that uses `fail()` to force backtracking. You may have already found out that values obtained at the bottom of the loop are quickly lost as Prolog backtracks past where these values were calculated. With the use of the database, you can store these values to be retrieved later when they are needed. Clever use of asserting values into the database, coupled with the `findall()` predicate, allows easy access to groups of data that have been obtained in a "failing" loop. Here is an example of how this might work:

```

Domains
    item = symbol
    cost = real
    quantity = integer
    total_worth = cost*

Database
    inventory_value(item, cost)

Predicates
    item ( item, cost, quantity )
    calculate_inventory_value
    calculate_total_inventory_worth ( total_worth, cost )

Clauses
    calculate_inventory_value :-
        item(Item, Cost, Quantity) ,
        Total_worth = Cost * Quantity ,
        writef("%-12 Qty: %-8 Cost Per: $%5.2f Total: $%8.2f.\n",
            Item, Quantity, Cost, Total_worth) ,
        assert( inventory_value(Item, Total_worth) ) ,
        fail.
    calculate_inventory_value :-
        findall(Worth, inventory_value(_,Worth), Worth_list) ,
        calculate_total_inventory_worth(Worth_list, Grand_total) ,
        writef("\n\nThe grand total is $%9.2f.\n", Grand_total).

    calculate_total_inventory_worth([],0) :- !.
    calculate_total_inventory_worth([H|T],Worth) :-
        calculate_total_inventory_worth(T,Worth1) ,
        Worth = H + Worth1.

    item(frame, 22.95, 400).
    item(wheels, 13.75, 800).
    item(handle_bars, 12.99, 350).
    item(seat, 5.92, 452).

GOAL
    makewindow(1,2,3," Cost of Goods in Inventory ",0,0,25,80) ,
    calculate_inventory_value.

```

In Prolog, there is no such thing as a global variable. If you plan to use a value in the body of a predicate, then that value must be passed into the predicate through the head of the rule. Because of this, it is sometimes necessary to carry around several values from clause to clause. An example is a grand total that is calculated in one part of a program but that is used in several processes in the program. Here it is necessary to carry around the total to all the processes that need to reference it.

The database can help out in this situation. It is possible to store values in the internal database that can be accessed from anywhere inside the program. In this way, the database can act as a

place to store global values. Once a value is calculated, just assert it into the database. Then when the value is needed it can be easily accessed through the database. When the database is used in this manner, it is no longer necessary to cart values all over the program, and the program becomes easier to maintain and read.

External Databases

Apart from internal databases, Turbo Prolog also supports external databases. External databases store Prolog terms in database *chains*. These chains may be indexed for quick data retrieval using *B+ trees*.

The advantage of using external databases becomes evident when you begin to write systems that use large amounts of data. If you try to store all your information in the internal database, you will quickly run out of room for your program to process the data. Moving the data to external files frees up the space that would otherwise be taking up valuable processing memory.

Database Chains

External database chains are similar to internal databases. Chains of terms are stored sequentially and are accessed one at a time. External databases are given a database name, and within each name separate chains of data can exist. While each separate chain can be manipulated as a separate group of data, the entire database group can also be easily handled as a single unit.

External chains store data in the form of Prolog *terms*. Remember that a term can be either a compound object (which has a structure much like a Prolog fact) or a value. Lists, compound structures, integers, reals, and strings are all valid terms that can be inserted into the external database. Since the external database can be used to store any type of Prolog term, the external database is slightly more flexible than Turbo Prolog's internal database.

Creating External Databases

Unlike internal databases, external databases are kept separate from the actual Prolog program. While they are easy to manipulate, accessing external databases involves more work than it does to access internal database facts. With an internal database, you just assert the fact and you are off and running.

If you are creating a new external database, you must use the predicate

```
db_create()
```

to initialize the database. Three arguments are used with `db_create()` to create a new external database: a database name, the database location, and a file name.

The first argument is a database name; it must be declared ahead of time in the Domains section of the program. A special domain (like the "file" domain used for declaring symbolic file names) is used to declare external database names. The domain used to declare the database names is `db_selector`. Here is an example of how to declare two external database names:

Domains

```
db_selector = external_db1 ; birthday_data
```

An external database can be located in one of three different places: in RAM memory, in a disk file, or in expanded memory specification (EMS) memory. Each of the three database locations has advantages and disadvantages, as Table 11-1 illustrates.

The location of the database is specified by giving it one of the predefined place names used by Turbo Prolog. These are `in_memory` for a RAM-based database, `in_ems` for an EMS-based database, and `in_file` for a DOS-file-based database.

The following two lines show how `db_create()` is used:

```
db_create(db_name, "DOS_FILE.DAT", in_file)
```

```
db_create(people_db, "people", in_memory)
```

The first of the two calls initializes a database named `db_name`.

Location of External Database	Advantages	Disadvantages
RAM memory	Fast access	Uses up memory (heap) that can otherwise be used for processing the program Size is limited to available heap space
DOS File	Size of database is limited only by size of disk (can be several megabytes on a hard disk)	Access to disk is relatively slow
EMS Memory	Fast access Doesn't utilize heap from program	Needs an Expanded Memory Board Size is limited to size of available extended memory

Table 11-1. *Advantages and Disadvantages of External Database Locations*

The database will be created on disk in a DOS file named DOS_FILE.DAT.

The second call creates an external database in RAM. This database will have a database name of "people_db", while its symbolic file name (similar to the symbolic file names given to other Turbo Prolog files) is "people". Databases created both in memory and in EMS are given symbolic file names when they are created, whereas an external database created on disk is given an actual DOS file name.

Opening and Closing External Databases

It is important to close external databases when you have finished writing to them. If you don't, the database will be invalid when you try to reopen it later. Attempting to use `db_open()` with a database

that has not been properly closed will yield an error, and your program will come grinding to a halt. Closing an external database is easily done with a call made to the predicate

`db__close()`

Taking a single database name as an argument, `db__close()` closes the database and marks it as valid so that it can be accessed later. In addition to marking a database as valid, `db__close()` also flushes any information that the database may have stored in a buffer.

Once a database has been created and stored, it can be reopened with the predicate

`db__open()`

This predicate is almost identical to `db__create()`, in that they both take the same three arguments. The only difference is that `db__create()` creates a new database, while `db__open()` reopens one that has already been created.

Inserting Terms into External Chains

External database chains are almost identical in function to Turbo Prolog's internal databases. Facts (or terms) are added to the database at runtime. Information in the database can be accessed by Prolog, and inferences can be drawn using the stored knowledge. To add a fact to the external database, one of the three `chain__insert` predicates must be used:

`chain__inserta()`

`chain__insertz()`

`chain__insertafter()`

You should feel comfortable with the predicates `chain__inserta()` and `chain__insertz()`, since they are identical in function to the built-in predicates `asserta()` and `assertz()`. Using these two predicates, inserting a fact into the external database takes five argu-

ments. The first of the five arguments takes the name given to the database in the `db_create()` call.

After giving the correct database name, the database chain is specified as the second argument. These first two arguments indicate exactly where the fact is to be placed.

The third and the fourth arguments describe the fact that is to be inserted. In the third argument, the domain that the fact belongs to is given. This domain can be any user-defined domain or even a standard domain type. It is up to the programmer, however, to make sure that the fact to be inserted has a corresponding domain that relates to it. The fourth argument is the actual term that is to be inserted.

The fifth argument supplied to the predicates `chain_inserta()` and `chain_insertz()` is an output argument, meaning that a free variable must be given in this position. When called, this argument will return a *database reference number*, a unique number that refers to the data item inserted. The value returned in the fifth argument belongs to a special system-defined domain type called "Ref."

The third `chain_insert` predicate is used to insert a fact in between other facts that may already be in the database. The predicate `chain_insertafter()` also takes five arguments, but the layout is slightly different. The difference between `chain_insertafter()` and the other two `chain-insert` predicates is that the third argument to `chain_insertafter()` takes a database reference number instead of a domain describing the data being inserted.

Shown below is a small example program that opens a database in a disk file, inserts two facts into the database chain `birthday_chain`, and then closes the database:

Domains

```
db_selector = birthday_file
person = p ( string, string )
date = d ( integer, integer, integer )
birthday = bday ( person, date )
```

GOAL

```
db_create(birthday_file, "BIRTHDAY.DBA", in_file) ,

chain_insertz(birthday_file ,    % Database name
              birthday_chain ,    % Database chain
              birthday ,          % domain of term to be inserted
              bday(p("Micki","McAuliffe"), d(9,25,45)) ,
              Ref) ,             % Reference number returned

chain_inserta(birthday_file, birthday_chain, birthday ,
              bday(p("Claudio","Guzman"), d(8,6,39)), _) ,

db_close(birthday_file).
```

The chain that a fact is to be inserted into does not have to be declared. When Turbo Prolog comes across a new chain name, a new chain is created. Because of this, you must be especially careful to spell the chains you reference correctly, or else you may create an unwanted database chain.

Retrieving Data from an External Database

Like the internal database, an external database fact (or term) can be matched when a call is made to that fact. Unlike internal database facts, however, external data cannot be called upon directly. Instead, you must use a specific predicate to look up the terms that are stored externally. The predicate used to look up this information is

`chain__terms()`

Accessing the data stored externally is much like accessing the internal database. The predicate `chain__terms()` is nondeterministic, meaning that through backtracking, `chain__terms()` will return all the values stored in a particular database chain.

The format of `chain__terms()` is exactly the same as the predicates `chain__inserta()` and `chain__insertz()`. Five arguments are used: a database name, a chain, the domain of the term being referenced, the term, and a reference number. The distinction between the chain-insert predicates and `chain__terms()` is that the last two arguments of `chain__terms()` return values. The values return a database term and its respective database reference number.

The example that follows shows how to open a previously created database file and read the terms from it. The file opened and read from is the database file created in the previous program.

Domains

```
db_selector = birthday_file
person = p ( string, string )
date = d ( integer, integer, integer )
birthday = bday ( person, date )
```



```

Predicates
  read_terms

Clauses
  read_terms :-
    chain_terms(birthday_file, birthday_chain, birthday, X, _) ,
    X = bday( p(First, Last), d(M,D,Y) ) ,
    writef("% %'s birthday is on %d/%d/%d.\n",First,Last,M,D,Y) ,
    fail ; true.

GOAL
  makewindow(1,2,3," Birthdays ",0,0,25,80) ,
  db_open(birthday_file, "BIRTHDAY.DBA", in_file) ,
  read_terms ,
  db_close(birthday_file).

```

External Database Handling Made Easy

Often you will want to use external databases to store dynamic data, much as you do with internal databases. The greater flexibility and placement of the external database are strong features, but the unwieldy predicates makes this type of database seem cumbersome. However, there are several techniques you can employ to make the external database as easy to handle as the internal databases described earlier in this chapter.

For instance, most of the time when you insert a term into the database you will be using the same database name, chain, and domain; only the term you are inserting is different. Likewise, when you retrieve information out of the external database, you are only interested in the actual term. The database name, chain, and domain of the term you are looking for will most often be the same for all the terms you are looking for. Because of this, many of the external database predicates can be generalized to work just like the predicates of the internal database.

In the previous example, `chain__terms` is used to look up data stored in the external database:

```
chain__terms(birthday__file, birthday__chain, birthday, X, __) ,
```

While in a small program it would not be much of a problem to use this predicate call, in larger programs you will get bogged down when you need to make several calls to the external database.

Looking up external database terms can be greatly simplified by using a "look-up predicate" to do all the dirty work:

```
db_bday(First, Last, Date) :-
    chain_terms(birthday_file, birthday_chain, birthday, X, _),
    X = bday(p(First, Last), Date).
```

Now, anytime you want to look up a person's birthday, all you need to do is make a call to `db_bday()`. Also, if database reference numbers are needed, it is easy enough to add one more argument to the predicate `db_bday()`, and you're on your way.

Many of the external database predicates can be simplified in the manner shown above. The need to do so, however, depends on how often certain external database predicates need to be called. If you find yourself typing the same call over and over, with only one argument changing each time, it might be worth creating a look-up predicate for the larger predicate being called.

As an exercise, modify the example given in the first listing in this chapter so that it works with external database chains. In doing so, model as many external database predicates as you want to, so that they will work with look-up predicates. Using a look-up predicate can ease both the writing and reading of a program.

Updating External Database Terms

Aside from opening, closing, and adding data to external databases, you can also update data that has been placed in the database chain. The two predicates

```
term_replace()
```

and

```
term_delete()
```

will each work with a term that has previously been added to the

database. Both predicates use a database reference number that corresponds to the term that is either to be updated or deleted.

The predicate `term_delete()` simply takes a database name, the chain the term belongs to, and the reference number of the term to be deleted. To replace a term in the database, you give a database name, the chain, the reference number, and the new term that is to replace the old term.

Miscellaneous Database Predicates

Besides the basic external database predicates introduced here, Turbo Prolog provides several other built-in predicates to manipulate external database chains. Entire chains can be deleted with a single call, terms can be retrieved by reference number alone, external databases can be copied from one location to another, and complete statistics can be obtained concerning a particular named database. The *Turbo Prolog User's Guide* contains a complete reference of built-in external database predicates.

B+ Trees

In order to quickly look up information stored in external chains, Turbo Prolog supports an indexing method called B+ trees. This indexing scheme allows Prolog to quickly come up with a database reference number based on a key that you supply. After a database reference number has been obtained, accessing the correct term in the database chain is direct and easy.

When you create an external database chain it is possible to keep track of where you added the terms with the use of a B+ tree. In a B+ tree, a *key* is associated with each reference number in the database chain. Since each reference number refers to a unique database entry, finding the correct key will quickly lead you to the data item you are searching for.

Creating a B+ Tree

When you create or open a database chain, you must also create or open a B+ tree to keep track of the terms in the chain. Each time you modify a database chain, the B+ tree associated with that chain must also be updated so that the tree can properly reflect the database chain.

The predicate used to create the B+ tree is

```
bt_create()
```

This predicate takes five arguments, as follows:

```
bt_create( Database__name ,  
           B+__tree__name ,  
           B+__tree__selector ,  
           Key__length ,  
           Node__length )
```

The Database__name is a name like the one used in the database chaining predicates shown above. You must use the same database name for the database chain and the B+ tree that keeps track of the chain. This is because the B+ tree is actually stored in the same database file as the chains the B+ tree is indexing.

The B+__tree__name is a name you give to the B+ tree. This name is a string, and is similar to the name you give to external databases that store data. The third argument is an output argument, and belongs to the special domain bt__selector. When bt_create() is called, the third argument returns the B+__tree__selector, which is used to identify the B+ tree.

Key Length and Node Length The last two arguments used in bt_create() define the different sizes that the B+ tree will operate with. The fourth argument describes Key__length. Keys are string arguments and must be large enough to allow database items to be uniquely identified. It is important to dictate a key length large enough to hold the largest key. On the other hand, you won't want a key length to take up too much space, or it will require too much memory when used in conjunction with large databases.

B+ Tree Structure

The last argument to `bt_create()` requires that you have a basic understanding of how B+ trees operate. A B+ tree is a data structure that is broken up into separate *nodes* or *pages*. Each node to the tree has two nodes lower than itself, unless the node is the last one in succession. A node at the end of the tree is termed a *leaf node*. Leaf nodes do not have any nodes lower than themselves.

Each node on a B+ tree contains a group of keys within a specific range. Since no two nodes contain the same key values, a quick check can be made to see if the key being searched for lies within the range of a particular node. If so, the reference number to that key can be quickly retrieved.

The node that is below and to the left of a particular node contains keys that are less in value than the keys on the present node. Nodes that are below and to the right contain keys that are greater in value than the present node. If the key being searched for is not on the present node, then Prolog will make a comparison with the values held on that node. If the value is less than the values on the present node, then the left-lower branch will be searched. If the value is greater, the right-lower node will be examined for a match. This indexing method makes it possible to quickly determine which node a particular key is on and retrieve the reference number needed.

The "Node_length" argument in the `bt_create()` call determines the number of keys that are stored on each node of the tree. Both the key length and the node length arguments are integer values, and you may specify any values you wish to use. However, to create the fastest searching tree, you will have to experiment with these two values for each tree that you create. To start with, a node length of 4 is recommended for medium-size databases.

Opening and Closing B+ Trees

Like database chains, B+ trees can be closed and reopened with the predicates

`bt_open()`

`bt_close()`

To open a B+ tree that has previously been closed, you must supply the predicate `bt_open()` with the database name and the B+ tree name. The predicate will return the B+ tree selector in the third argument. To properly store a B+ tree, you must close it with `bt_close()`. This predicate takes two arguments; the first is the database name and the second is the B+ tree selector that has been assigned to that B+ tree.

The following example program shows how to see whether an external database has been created. If it has, then the database, along with its respective B+ tree, is opened. If it hasn't, then the database and B+ tree are created.

```
external_open(Bt_sel) :-
    existfile("PARTS.DBA"), !,
    db_open(parts, "PARTS.DBA", in_file) ,
    bt_open(parts, "Parts_tree", Bt_sel).

external_open(Bt_sel) :-
    db_create(parts, "PARTS.DBA", in_file) ,
    bt_create(parts, "Parts_tree", Bt_sel, 8, 4).
```

When closing a set of database chains and B+ trees, you must first close the B+ tree before closing the respective database file. Doing so ensures that the buffers for the B+ tree are properly flushed to the file before it is closed.

Updating a Tree

Like the other databases, B+ trees can be updated by adding and deleting keys. Indexing the keys is automatically kept up by the Prolog system. The two predicates used for updating a B+ tree are

`key_insert()`

and

`key_delete()`

Both predicates take the same arguments, and will either add a key and reference number or delete the key and reference number given. Each predicate takes four arguments, which must all be bound when the predicate is called. The first two arguments are the database name and the B+ tree selector. The third and fourth arguments are the key to insert or delete and the associated reference number.

Searching for a Key

To search for a given reference number, which leads to the data item you are looking for, you must use the predicate

`key_search()`

Searching for a key takes the same four arguments that it takes to insert or delete a key from the B+ tree: a database name, a B+ tree selector, a key, and a reference number. The only difference is that the reference number is used as an output argument because Prolog returns the number of the key that you are looking up.

If the key cannot be found, then `key_search()` will fail. Also, if duplicate keys exist in the B+ tree, `key_search()` will return just one of the reference numbers belonging to that key. Using the predicates

`key_next()`

and

`key_prev()`

you can search for other terms in the database that match the key you are looking for.

Using B+ Trees

The last program in this chapter is an example showing how B+ trees are used in association with database chains. The program is set up to work with the parts database, similar to the bicycle database introduced in Chapter 9.

The program prompts for a major part, then its subparts. The parts and subparts are then stored in the external database, with the major component being the key field for the B+ tree. After all the parts have been added, the program prompts the user for a major component. This part is looked up and displayed, along with its respective subparts.

Domains

```
db_selector = parts
sub_parts = part*
part = string
component = p(part, sub_parts)
```

Predicates

```
external_open ( bt_selector )
external_close ( bt_selector )

enter_parts ( bt_selector )
enter_subparts ( sub_parts )
insert_part ( part, sub_parts, bt_selector )

look_up_parts ( bt_selector )
read_part ( part, bt_selector )

list_parts ( sub_parts )
list_more
```

Clauses

```
/* * * * * *
 * Database Opening Routine *
 * * * * * */

external_open(Bt_sel) :-
    existfile("PARTS.DBA"), !,
    db_open(parts, "PARTS.DBA", in_file) ,
    bt_open(parts, "Parts_tree", Bt_sel).
external_open(Bt_sel) :-
    db_create(parts, "PARTS.DBA", in_file) ,
    bt_create(parts, "Parts_tree", Bt_sel, 8, 4).

external_close(Bt_sel) :-
    bt_close(parts, BT_sel) ,
    db_close(parts).

/* * * * * *
 * Insert Parts Into the Database *
 * * * * * */

enter_parts(Bt_sel) :-
    clearwindow ,
```

```

write("Enter a part (press RETURN to quit): ") ,
readln(Part), nl ,
Part <> "" , ! ,
enter_subparts(Sub_parts) ,
insert_part(Part, Sub_parts, Bt_sel) ,
enter_parts(Bt_sel).
enter_parts(_) :- clearwindow.

enter_subparts([Subpart|Parts]) :-
write("Enter a subpart (press RETURN to quit): ") ,
readln(Subpart), nl ,
Subpart <> "" , ! ,
enter_subparts(Parts).
enter_subparts([]).

insert_part(Part, Subparts, Bt_sel) :-
chain_insertz(parts, part_chain, component,
              p(Part, Subparts), Ref) ,
key_insert(parts, Bt_sel, Part, Ref).

/* * * * * *
* Look Up Parts *
* * * * * */

look_up_parts(Bt_sel) :-
clearwindow ,
write("Enter the part you wish to look up: ") ,
readln(Part), nl ,
Part <> "" ,
read_part(Part, Bt_sel), ! ,
look_up_parts(Bt_sel).
look_up_parts(_).

read_part(Key, Bt_sel) :-
key_search(parts, Bt_sel, Key, Ref), ! ,
ref_term(parts, component, Ref, p(Part, Subparts)) ,
writef("The part \"%s\" has the following subparts:\n" ,
      Part) ,
list_parts(Subparts) ,
list_more.
read_part(_, _) :-
write("The part was not found.\n"), beep ,
list_more.

/* * * * * *
* List Parts in Database *
* * * * * */

list_parts([]) :- !.
list_parts([H|T]) :-
write(H), nl ,
list_parts(T).

list_more :-
write("\nDo you wish to see another part? ") ,
readchar(Ans) ,
upper_lower(Ans, 'n'),
!, Fail ; true.

GOAL
makewindow(1,2,3," Parts Database ",0,0,25,80) ,
external_open(Bt_sel) ,
enter_parts(Bt_sel) ,
look_up_parts(Bt_sel) ,
external_close(Bt_sel).

```

In small database applications, keeping a B+ index may be more trouble than it's worth. But when larger files are involved, B+ trees are indispensable for quick look-ups and data sequencing. Since multiple keys can be kept for a single database chain, creating a relational database is greatly simplified with the use of B+ trees.

A complete guide to all the external database predicates and their uses could be the topic of a book as large as this one. This chapter has introduced only the major points that will get you started. Take the time to browse through the Turbo Prolog manuals for more information on how databases can be put to work for you.

Part Four

You have now been introduced to the basics of programming in Turbo Prolog. By now you are probably ready to attack more challenging programs and use your new skills on more complex problems. The last section of this book is devoted to advanced programming techniques and shows the advanced features found in the compiler. This section also introduces you to the Turbo Prolog Toolbox, and gets you started with the Borland Graphics Interface (known as the BGI). The last chapter in the section provides an advanced programming application, making use of many of the tools and techniques you have learned throughout this book. Altogether, this section will give you a good foundation for continuing to develop advanced Turbo Prolog applications of your own.

12 *Advanced Techniques*

By now, you should feel fairly comfortable with the basic skills required to write simple Prolog programs. With a little imagination and ingenuity, you will be able to create expert systems, schedule programs, and program small database applications.

This chapter discusses two styles of techniques and concepts required to develop large programs. The first style deals with advanced functions built into the Turbo Prolog compiler itself. Sophisticated debugging features, compiler directives, and the use of program modules all are options that aid in the creation of large programs. The second style concerns programming strategies that can be used to attack the complex problems that often crop up.

Advanced Compiler Features

As the applications you write get larger and more complex, you will at times encounter problems that need to be approached with care. Sometimes, the easiest way to conquer these problems is to let the compiler reduce the problem for you, paving the way for an

easy solution. Using the advanced features built into Turbo Prolog, difficult problems can be taken on and solved with a minimum of trouble.

Compiler Directives

The first set of advanced compiler features comes in the form of *compiler directives*. Compiler directives are separated into groups, one for debugging programs and one for easing the task of creating large programs. Directives are placed at the top of a program, before any code declarations, and instruct the compiler to turn certain features on or off.

You may not have realized it, but you have already used one of the most important compiler directives in your programs. When you turn trace on, you are actually enabling the tracing mechanism with a directive. The word *trace* is a compiler directive and, like all directives, is placed at the very beginning of your program code. Table 12-1 shows a complete list of the directives that can be used with Turbo Prolog. As this chapter progresses, the directives will be introduced and you will be shown how they are placed and used.

Advanced Tracing Techniques

Prolog refers to four *goal ports* when outputting a program trace: CALL, RETURN, REDO, and FAIL. During the processing of a program, Prolog makes a CALL when each new subgoal is reached. After a RETURN is made, proving the CALL to be true, the next subgoal in line is then CALLED. In this fashion, the program moves forward from one subgoal to the next. When a subgoal cannot make a RETURN from a CALL, that CALL is said to FAIL. FAIL forces Prolog into backtracking, which leads to a REDO of the last nondeterministic predicate passed. The port REDO is similar to CALL, except that REDO is used when the processing of a program is in retreat.

When viewed like this, you can see that the RETURN port allows processing to continue forward, while FAIL causes Prolog to turn back and look for alternative solutions. Prolog makes CALLs as predicates succeed (forward processing), and makes REDOs when predicates fail (backward processing).

Directive	Use
<code>bgidriver</code>	Loads in a specific BGI graphics driver, to be linked in at compile time
<code>bgifont</code>	Loads in a specific BGI graphics font, to be linked in at compile time
<code>check__determ</code> <code>code</code>	Flags predicate definitions that are nondeterministic in nature Sets up the size of the compiled code array. Size is set in 16-byte paragraphs; for example, <code>code = 2000</code>
<code>config</code>	Reads in a named setup file (such as <code>PROLOG.SYS</code>), setting up certain defaults for a compiled program to run with. Examples of settings that are affected are the color of help menus and the color of the auxiliary edit windows.
<code>diagnostics</code>	Compiles a report based on the predicate definitions in the program. Report on predicates includes the predicate size, deterministic nature, flow patterns, argument domains, and whether the predicate is global or local
<code>errorlevel</code>	Sets up the DOS error level, overriding the environment setting
<code>heap</code>	Sets up amount of heap space that the compiled .EXE program will take up. If set to 0, or not set, the program will take up all available RAM. Size is set in 16-byte paragraphs; for example, <code>heap = 4000</code>
<code>include</code>	Reads a section of Turbo Prolog code into the position indicated by the directive
<code>nobreak</code>	Prevents a program from terminating by pressing CTRL-BREAK
<code>nowarnings</code>	Suppresses certain compiler warnings, such as "variable is only used once"
<code>prntermenu</code>	When this directive is placed in the code, it will allow the Printer status menu (ALT-P) to be accessed from the compiled program
<code>project</code>	Declares a program module to be a part of the named program project
<code>shorttrace</code>	Turns on the compiler's tracing mechanism, retaining the compiler's optimizations
<code>stack</code>	Sets up the program's stack size, overriding the size set up by the environment. Size is set in 16-byte paragraphs; for example, <code>stack = 2000</code>
<code>trace</code>	Turns on the compiler's tracing mechanism, and disables the compiler optimizations
<code>trail</code>	Sets up the program's trail size, overriding the size set by the environment. Size is in 16-byte paragraphs; for example, <code>trail = 100</code>

Table 12-1. *Compiler Directives*

Turbo Prolog offers two separate *modes* that allow you to inspect the processing that goes through these ports: *trace* and *shorttrace*. In addition to these two Trace modes, you can elect to *spy* on the predicates that interest you, or you can turn tracing on and off to inspect different sections of your program code.

Trace and Shorttrace The two compiler directives *trace* and *shorttrace* perform essentially the same function. The only difference is that *shorttrace* employs Turbo Prolog's optimizations in its trace output, while *trace* does not.

When running programs outside of Trace mode, Turbo Prolog uses a good deal of optimization to increase program execution speed. Most notable is the *tail-recursion elimination* performed on tail-recursive predicates.

When a recursive predicate is encountered during processing, each call made in the recursion cycle needs to return so that the program can continue forward. When a predicate is defined as tail recursive, Turbo Prolog can optimize the recursive cycle by jumping back to the beginning of the recursive process once the first return has been made. Jumping back increases the speed of a program because Prolog doesn't need to spend time returning from each call made inside the recursive loop. This means that each time a recursive call is made, Prolog doesn't need to physically generate a return for the program to continue forward.

The effect of this optimization can be seen by using *trace* and *shorttrace* in the program that follows. When *trace* is used, you get a verbose listing of all the RETURNS made from each CALL, even in the recursive predicates. This is because Prolog's optimizations are not in effect. When you don't want to get bogged down with this detailed listing, use *shorttrace* to enable Prolog's optimizations and shorten the output of the trace listing.

```

trace
Domains
    list = symbol*

Predicates
    make_list ( list )

```

Clauses

```
make_list([Head|Tail]) :-
    readln(Head) ,
    Head <> "" , ! ,
    make_list(Tail).
make_list([]).
```

GOAL

```
makewindow(1,2,3," Make A List ",0,0,25,80) ,
write("Enter the elements to make up a list.\n" ,
      " (Enter a blank line to terminate the list)\n\n") ,
make_list(List) ,
clearwindow ,
write(List).
```

Spying on Predicates You may sometimes find that most of your program works, but certain sections of your code do not produce the solutions you desire. It may be tedious to trace through your entire program just to track down a bug in one or two predicates. However, you can spy on specific predicates by declaring the predicates you want to trace. Since only the predicates listed are traced, it becomes easy to track the problem down. For example, suppose you have the following program:

Domains

```
cust = c(first, last)
order = o(part, quant)
first, last = string
id, part = symbol
quant = integer
```

Database - customers

```
customer ( cust, id )
```

Database - orders

```
order ( id, order )
```

Predicates

```
print_orders
write_orders ( Id, Order )
```

Clauses

```
print_orders :-
    write("Customer ID      Part Ordered      Quantity\n" ,
          "=====      =====      =====\n") ,
    fail.
print_orders :-
    customer(_, Id) ,
    order(Id, Order) ,
    write_orders(Id, Order) ,
    fail ; true.
```

```

write_orders(Id, o(Quantity,Part) ) :-
    writef(" %6\t %10\t %4\n",Id,Part,Quantity).

/* Database of Customers */
customer(c("Alexandra", "Guzman"), xd0039).
customer(c("Tim", "Isom"), xd0039).
customer(c("Martha", "Wheeler"), xd0046).
customer(c("David", "Walzer"), xd9908).
customer(c("Bill", "Weeber"), xd1334).

/* Database of Orders */
order(xd0046, o(fan,2)).
order(xd9908, o(d_washers,5)).
order(xd0046, o(o_rings,12)).
order(xd1334, o(pipe_wrench,1)).
order(xd9908, o(tube_cutter,8)).
order(xd0039, o(stoppers, 15)).

GOAL
    makewindow(1,2,3," Orders ",0,0,25,80) ,
    print_orders.

```

You can tell that everything is working fine, except for the section that outputs customers' orders. Placing the trace directive

```
trace write__orders
```

at the top of the code tells Prolog that you only want to see the calls and returns made to the predicate `write__orders()`.

Using this same technique, you can order several predicates to be spied on. To do this, just list the predicate names after the trace (or `shorttrace`) compiler directive, separating each one with a comma, such as

```
trace write__orders, print__orders
```

Tracing Sections of Code Spying on predicates can be useful, but often it is difficult to know in advance exactly which predicates are presenting a problem. Perhaps you want to get a full trace of a particular section of code, but you don't want to spend the time pressing F10 to single-step through the code that you already know works. The built-in predicate

```
trace()
```

is handy because it allows you to specify exactly which sections of code you want to trace. The predicate takes a single argument (the symbols on and off) and turns the tracing either on or off. Notice that before you can turn tracing on or off in this manner, one of the trace compiler directives must be placed at the top of the code. After the directive is in place, supply `trace()` with an argument of either on or off, and you have full control of which code is to be inspected and which code is to be left unwatched.

`trace`

`Predicates`

```
parent ( symbol, symbol )
ancestor ( symbol, symbol )
any_ancestors ( symbol )
find_ancestors ( symbol )
```

`Clauses`

```
any_ancestors(Person) :-
    parent(_, Person), ! ,
    clearwindow ,
    find_ancestors(Person).
any_ancestors(_) :-
    write("\nI'm sorry, but I don't know about that person!\n").

find_ancestors(Person) :-
    trace(on) ,
    ancestor(Who, Person) ,
    write(Who," is the ancestor of ",Person,".\n") ,
    fail ; trace(off).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :-
    parent(X,Z),
    ancestor(Z,Y).
```

```
/* Parents Database */
parent(martha, kenny).
parent(jim, suzy).
parent(jim,martha).
parent(kenny, sam).
parent(kenny, sally).
```

`GOAL`

```
trace(off) ,
makewindow(1,2,3," Ancestors ",0,0,25,80) ,
write("Enter the name of a person and " ,
      " I'll write out their ancestors!\n      ") ,
readln(Person) ,
any_ancestors(Person) ,
write("\nThat's it!").
```


Files	Edit	Run	Compile	Options	Setup
Editor				Dialog	
Error Correction	Line 47	Col 31	C:.\TEMP\FOR		
Diagnostics					
DIAGNOSTICS FOR MODULE: C:.\TEMP\FOR_NEXT.PRO					
Predicate Name	Type	Determ	Size	Domains -- flowpattern	
goal	local	YES	91	—	
get_customer_id	global	YES	73	person,id -- i,o	
fact	dbase	NO	98	integer,integer -- o,o	
for	local	NO	58	integer,integer,integer -- i,i,o	
next	local	YES	23	integer,integer -- i,i	
assert_facts	local	NO	189	—	
write_facts	local	YES	139	—	
foo	NOT USED			integer	
Total size			671		
Press the SPACE bar					

F2-Save F3-Load F6-Switch F9-Compile Alt-X-Exit

Figure 12-1. A sample diagnostics report

Tracing Tools

Besides examining the flow through your program, Turbo Prolog offers several other tools to aid in debugging code. In addition to tracing, compiler directives can help with your debugging sessions. You also have the capability of capturing the output of your trace either to a file or to a printout.

Diagnostics The compiler directive *diagnostics* can produce a report detailing the statistics of all your user-defined predicates. As Figure 12-1 shows, the report names which predicates are used and which are not, the flow patterns of the predicates that are called, and whether the predicates are deterministic or nondeterministic. The diagnostics report also indicates how the predicate was declared (if it is global or local, or if it was declared in the Database section of your program) and the size of each predicate. To get this report, just place the directive *diagnostics* at the top of your program. After this, a report will be generated when you compile the code.

Trace and Log Status Menus To allow for an even more flexible debugging environment, Turbo Prolog offers two status menus that supply debugging functions. The Trace Status and Log Status menus each perform a different task, giving you full control of how, when, and where tracing takes place.

The Trace Status menu is available only when you are actually tracing a program. Once you have started to run a program in Trace mode, you can activate the Trace Status menu by pressing the hot-key sequence ALT-T. A menu with three choices will pop up, allowing you to toggle the tracing options on or off. To use the Trace menu, just select an option and press ENTER to toggle the option either on or off. After setting the options to your liking, press F10 once or twice to return you back to the program run.

The first of the three options allows you to turn tracing on or off, and is much like placing the `trace()` predicate inside your program code. This is a handy, interactive way of skipping over certain sections of the program that don't need tracing and turning tracing on for the critical sections of your program.

The second option in the Trace menu gives you the ability to turn the Trace window on and off. When the Trace window is inactive, the flow through the goal ports does not go to the Trace window. Keeping the Editor window active allows you to view the order of the calls made to the clauses as you single-step through your program.

The last option allows you to turn the Editor window on or off. When the Editor window is off, you do not need to press F10 at each step of processing. Instead, the processing through the goal ports goes to the Trace window without waiting for you to press F10 at each call. Notice that turning both Editor and Trace windows off effectively turns off the tracing of your program.

The second status menu allows you to keep a log of your tracing activity. By pressing ALT-P to bring up the Log Status menu, you can direct the screen output either to your printer or to a DOS file. The first option of the Log Status menu allows you to toggle your printer on and off. When the printer status is toggled on, all characters output to the screen are echoed to the printer. The second option allows the same report to be given, but instead of sending it to the printer, the report is sent to a file titled PROLOG.LOG. (When you are done, to properly close the file be sure to turn this

option off.) This feature also allows you to save the report generated by the diagnostics compiler directive out to a file.

Using the two status menus together can greatly increase the productivity of your debugging time. By turning the Editor window off and turning the printer on, you can quickly get a printout of the trace report. One word of caution: Be careful when using this technique to trace large programs. You may be surprised to find yourself wading through reams of trace printout!

As an exercise, try this technique while tracing the procedure that defines the predicate `reverse()`. If you set up your system correctly, the goal

```
reverse([1,2,3,4], X).
```

should produce a printed listing of the processing through the goal ports of your program. After you successfully get a printed listing of the trace, take a pencil and match each `CALL` in the program with its respective `RETURN`. Doing this exercise using both trace and shorttrace will give you a good idea of the functionality and differences between these two directives.

```
Domains
  integerlist = integer*

Predicates
  reverse ( integerlist, integerlist )
  append ( integerlist, integerlist, integerlist )

Clauses
  reverse([], []) :- !.
  reverse([H|T1], New_list) :-
    reverse(T1, T2),
    append(T2, [H], New_list).

  append([], L, L).
  append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Include Directive

The *include* compiler directive allows you to clarify your program by letting you bring in sections of code from other Prolog files. The files included can be entire Prolog procedures with all the declarations or they can just be a set of clauses that you want to include in your program.

With the include directive, you can create small files containing groups of predicates that address specific tasks. For instance, let's say that you have developed several useful predicates that manipulate lists. If you have a program that makes use of these procedures, you will certainly not want to spend time rewriting all the predicates you have already written. In addition, you may not want to place the entire list of predicates in your code because your program would then appear cluttered.

The include directive is available for just this type of situation. Unlike other directives, the include directive can be placed anywhere inside your program code—there is no restriction dictating where this directive can be placed and it may be used several times in one program. The include directive reads a file with a DOS filename into the program at compile time. The reason for this is to clear up your code, at the same time allowing you to create files of Prolog predicates that can be included in any of your programs. Here is an example of its use:

```
include "C:\PRO_FILES\LIST_PRD.PRO"
```

When this line of code is listed in your program, the file LIST_PRD.PRO will be read into the program at compile time, placing the code where the include directive is located.

The only restriction is that the file to be included must begin with a Turbo Prolog section keyword (such as Clauses, Domains, or Predicates). In this way, your program may be made up of several sections of code, with each section containing its own Domains, Predicates, and Clauses sections. Since there can be more than one Clauses, Predicates, or Domains section in your code, you must be sure that the declaration of a domain or predicate precedes the use of that domain or predicate in the body of your code. In other words, a call to a predicate cannot come before that predicate has been declared in the code.

Creating Projects

When you begin to create programs that are too large to compile as single files, you will need to break your programs up into separate modules. Appropriately named, this technique is called *modular programming*.

As you keep adding code to a large program, Turbo Prolog will let you know when your program has become too large to compile as a single file. If you attempt to compile a file that is too large, an error message will appear stating that you have run out of available heap space. Once this happens, you have several alternatives to choose from to rescue your program.

First, you can try to compile the file to an object file, then link it from the DOS command line to form an executable program. While this may solve the problem temporarily, it will not work for long if you still have more to add to your program. Another option is to try to decrease the stack size, thus freeing up more heap space, but this too is only a temporary fix.

Ultimately, you will need to break your program file up into separate modules and compile your program as a *project*. This may sound complicated, and the fear of breaking your program up may cause you to put this step off until the last minute, but fear not. Modular programming is not difficult. Once you have taken the step to create a project, you need not worry about the size your program grows to. The sky (or just about) is the limit!

The easiest way to create a modular program is to plan for it in advance. If you have a large program to write, outline your program as a project before you even start coding. Nevertheless, sometimes you'll be caught off guard and need to break a program down without having planned for it. If this happens, remember that the first project is always the toughest, mostly because of the mental block involved with breaking up your beloved program.

Global Predicates and Domains The purpose of a project is to create separate files that are simply small pieces of the main program. These files are compiled as different units, but together they make up one large program. Once these files have been compiled (to object files), they can be linked together to form a final executable file (a file with an EXE extension). The actual code is the same as for a program that is compiled from a single file, except that the predicates are spread out through separate files.

Predicates that are defined in one module but are called from another are called *global predicates* and must be declared in the Global Predicates section of your program. In addition, global predicates that make use of special user-defined domains must have those domains declared in the Global Domains section.

When a predicate is declared as a global predicate, you must supply the flow pattern(s) that the predicate is to be used with. For example, if you are going to declare the predicate `append()` as global, you will have to figure out the different ways the predicate is going to be called. You will have to decide in advance which arguments will be bound and which will be free when the predicate is used. The following predicate declaration shows how `append()` is declared to be global, allowing two different flow patterns to be used:

```
Global Domains
  list = integer*

Global Predicates
  append(list, list, list) - (i,i,o) , (i,o,i)
```

When `append()` is declared like this, you can call it in one of two different forms. First, `append()` can be called with its normal form of concatenating two lists together, as is shown with the first flow pattern of (i,i,o). Or, second, `append()` can be called to see if a list makes up the first portion of a larger list. Using the second flow pattern given for `append()`, a call could look like this:

```
append([1,2,3], —, [1,2,3,4,5,6])
```

If this call is made to `append()`, the first and third arguments are bound and the second argument is free. When `append()` is used in this form, it will succeed only if the first argument matches the first part of the third list, as in the call just shown.

Partially Bound Arguments When you create a modular program, keep in mind that it is not possible to call a global predicate when one of its arguments is partially bound. For instance, let's say that you want to declare a predicate that takes a compound object as an argument:

```
Global Domains
  customer = p(first, last, id)
  id, first, last = symbol

Global Predicates
  get_customer_name ( person ) - (o)
```

If this predicate is declared as global, you cannot call `get_customer_name()` like so:

```
get_customer_name( p(First, Last, p39984) )
```

The argument to this call is partially bound (has both input and output values inside of a single argument) and is not permitted as a global call. Because of this restriction, it is best to write most compound-object and list-handling procedures inside of a single project module.

The key to keeping your projects easy to code and easy to maintain is to keep global predicates to a minimum. This is why planning ahead can be advantageous. When laying out your program, try to envision it in isolated units. Ideally, each unit will have a single call leading into it. If this can be achieved, you will need only one global predicate for each individual module because a single call will spark an entire module into action.

Defining the Project Module Each project has a *project module*. The project module lists all the separate program modules that are contained in the project. If your project has only two modules, the project module will contain only two entries.

The easiest way to create the project module is to use the Edit PRJ File selection found under the Options main menu choice. When this menu item is chosen, you will be prompted for a module name. Enter the name of the new module you wish to create, or supply the name of a module you wish to edit. In either case, you will be placed in the editor, ready to edit the project file. If you are creating a new project file, you will be provided with a clean slate. Type the name of each module file in the project on a separate line, and finish the project module by pressing F10. Turbo Prolog will automatically save the project file to the current directory and append the file extension .PRJ onto the end of the file name.

Writing Program Modules A project is not much different than any other compiled program. The major difference is that in modular programming predicates are spread out through several modules or files. Like other code that you compile into a finished, executable program, a project must contain a goal. In a project only one module is allowed to have a goal; that module is called the *main module*. Modules other than the main module must not contain a Goal section.

The one thing that separates a module source file from a non-module file is the presence of the compiler directive *project*. The project directive must be the first line of code in each module, and all modules in a project must contain the *project declaration*. The project declaration consists of the compiler directive *project* followed by the project file name enclosed in quotes. Here is an example:

```
project "my__proj"
```

This declaration defines the file to be a part of the project "my__proj."

Global Databases Often you will want to include databases in your projects. Turbo Prolog provides a graceful way to incorporate databases into your modules by allowing both local and global databases. Local databases consist of facts that can be accessed from within only one module. Global database facts, on the other hand, can be accessed across module boundaries, hence the name global. To declare a set of database facts as global, use the Global Database section to declare the database predicates.

Global Include File Since each module in a project will have the exact same global declarations (global domains, databases, and predicates), it is easiest to incorporate these declarations into the program modules by using the include compiler directive to include all of your global declarations. Instead of typing the same declaration into each project module, create a file (often called GLOBALS.PRO), and include this file in each module. NOTE: Global declarations must precede local declarations, so be sure that the file GLOBALS.PRO is included before any local declarations are made. Here is an example showing how to create a project. It is broken up into several files, or modules, the first of which is the main module, entitled "MY PROJ."

```
/* MY_PROJ.PRO */  
  
/*  
 * This is the main module of the project.  
 */
```

```

project "MY_PROJ"

include "GLOBALS.PRO"

Predicates      % Local Predicates
consult_orders
main_menu
menu ( integer )
process_choice ( integer )
get_choice ( integer )

Clauses
main_menu :-
    makewindow(1,2,3," Client Orders ",0,0,25,80) ,
    repeat ,
    menu(Choice) ,
    process_choice(Choice) ,
    Choice >= 3.

menu(Choice) :-
    makewindow(2,4,5," Main Menu ",5,7,9,20) ,
    write("(1) Add an order\n" ,
           "(2) List orders\n" ,
           "(3) Exit\n\n" ,
           "Enter your\nselection : ") ,
    get_choice(Choice) ,
    removewindow.

get_choice(Choice) :-
    readchar(C) ,
    Choice = C - '0' ,
    Choice <= 3,
    !.
get_choice(Choice) :-
    beep ,
    get_choice(Choice).

process_choice(1) :-
    !, add_order.
process_choice(2) :-
    !, list_orders.
process_choice(_).

consult_orders :-
    existfile("orders.dba"), ! ,
    consult("orders.dba", orders) ;
    true.

repeat. repeat :- repeat.

GOAL      % The goal is always in the main module.
consult_orders ,
main_menu.

```

Listed below is the ORDER module:

```

/* ORDER.PRO */

/*
 * This is the module which adds the orders.
 */

project "MY_PROJ"

include "GLOBALS.PRO"

```


Database - clients

```
client ( client, id )
```

Predicates

```
consult_clients
```

```
get_client_id ( client, id )
```

```
get_id ( id )
```

```
place_order ( Client, Id )
```

```
get_order ( item, quantity )
```

Clauses

```
add_order :-
```

```
    consult_clients ,
```

```
        makewindow(2,5,2," Client Name ",3,3,8,45) ,
```

```
        repeat, clearwindow ,
```

```
    write("Please enter the clients name.\n" ,
```

```
        "    First: " ) ,
```

```
    readln(First) ,
```

```
    write("    Last : " ) ,
```

```
    readln>Last) ,
```

```
    Client = c(First, Last) ,
```

```
    get_client_id(Client, Id) ,
```

```
    place_order(Client, Id) ,
```

```
    write("\nDo you want to add more orders (Y/N)? " ) ,
```

```
    readchar(Ans) ,
```

```
    upper_lower(Ans, 'n') ,
```

```
    save("orders.dba", orders) ,
```

```
    removewindow, !.
```

```
get_client_id(c("", ""), 0) :- !. % hit returns to exit
```

```
get_client_id(Client, Id) :-
```

```
    client(Client, Id), !. % Client is in the database
```

```
get_client_id(c(First, Last), Id) :-
```

```
    makewindow(13,5,4,"",5,5,7,42) ,
```

```
    writef("% % is not in the database.\n", First, Last) ,
```

```
    write("Would you like to add them as\n" ,
```

```
        " a client (Y/N)? " ) ,
```

```
    readchar(Ans) ,
```

```
    upper_lower(Ans, 'y'), ! ,
```

```
    write(Ans), nl ,
```

```
    get_id(Id) ,
```

```
    asserta( client(c(First,Last), Id) ) ,
```

```
    save("clients.dba", clients) ,
```

```
    removewindow.
```

```
get_client_id(,_ ) :- removewindow, fail.
```

```
get_id(Id) :-
```

```
    client(,_ , Max_id), ! ,
```

```
    Id = Max_id + 11 ;
```

```
    Id = 11. % First client
```

```
place_order(c("", ""), _ ) :- !.
```

```
place_order(Client, Id) :-
```

```
    makewindow(3,3,1," Place Order ",7,7,8,40) ,
```

```
    repeat, clearwindow ,
```

```
    get_order(Item, Quantity) ,
```

```
    assert( order(Client, Id, Item, Quantity) ) ,
```

```
    write("\nWould you like to add another\n" ,
```

```
        " order for this client (Y/N)? " ) ,
```

```
    readchar(Ans) ,
```

```
    upper_lower(Ans, 'n'), ! ,
```

```
    removewindow.
```

```
get_order(Item, Quantity) :-
```

```
    repeat, clearwindow ,
```

```
    write("Enter the item number: " ) ,
```

```
    readln(Item) ,
```



```

write("Quantity: ") ,
readint(Quantity).

consult_clients :-
    client(_,_), ! ; % if already consulted, do nothing
    existfile("clients.db") ,
    !, consult("clients.db", clients) ;
    true.

```

Listed next is the LIST module:

```

/* LIST.PRO */

/*
 * This is the module that prints out the orders.
 */

project "MY_PROJ"

include "globals.pro"

Clauses
    list_orders :-
        order(c(First,Last), Id, Item, Quant) ,
        writef("%-12 %-12 %5 %10 %6\n",First,Last,Id,Item,Quant) ,
        fail ;
        write("\nPress any key to continue...") ,
        readchar(_) ,
        clearwindow.

```

The next listing is the GLOBALS module:

```

/* GLOBALS.PRO */

/*
 * This is the Global Definitions file.
 */

Global Domains
    client = c(first, last)
    id, quantity = integer
    item, first, last = symbol

Global Database - orders
    order ( client, id, item, quantity )

Global Predicates
    client_order ( client, id ) - (i,o) , (o,i) , (o,o)
    list_orders
    add_order
    nondeterm repeat

```

Below is the PROJECT module:

```

/* MY_PROJ.PRJ */

/*
 * This is the Project module.
 */

my_proj
orders
list

```

Compiling a Project Once you have completed the project module and all of the program files, you are ready to compile the final program. This process is easily done from inside the Turbo Prolog environment. Turbo Prolog automatically compiles each module in the project to an object file, then links them together to form the final, executable program.

To compile your project, select "Compile" from the Main menu, then "P" for Project. You will be prompted for a project name, and like other Turbo Prolog prompts, you can either enter the project name or hit ENTER to get a selection of possible projects to compile.

Once you select a project to be compiled, Turbo Prolog will do the rest. All projects will be compiled to a final, DOS-executable file, and once the compiling and linking is finished you will be asked to run the program. If you select **Y**, your program will run; otherwise, you will be returned to the main menu. If you want to test the program after this, you will need to exit Prolog and run the program from the DOS command line, using the name of the project to start the program.

Since all projects end up as EXE files, there is no way to trace a program once it has been broken up into a project. For information on how to debug these programs, refer to the last section in this chapter, which is devoted to debugging large programs.

Coding Techniques

This section introduces techniques that you will be able to put to use in your own programs. The techniques are broken up into several subsections containing code that deals with counters, complex data types, negation, memory management, and optimizing your final product.

Looping and Counters

Several flow-control methods were introduced in Chapter 7. Among the structures described, the loop showed how a process could be repeated a given number of times. While counter loops are nor-

mally introduced fairly early in procedural language tutorials, these languages usually provide built-in structures to handle the looping.

In Prolog, counter loops must be modeled with the tools of the language. Although counter loops are not difficult, introducing these structures has been left until now because a fundamental knowledge of Prolog is needed to understand how these structures work.

In procedural languages, incrementing a counter variable can be easily done like this:

$$X = X + 1$$

However, in Prolog this logical construct will always fail (a value can never equal X and $X + 1$ at the same time). In Prolog, when a counter needs to be incremented, a new variable name must be created to hold the counter's incremented value, like this:

$$X1 = X + 1$$

This single difference between traditional programming languages and Prolog has led many seasoned programmers to throw Prolog aside, proclaiming it to be the language of a fanatic cult.

Recursive Versus Iterative Counters When you come across a situation that requires a counter to be implemented, you must decide which type of counter loop will work best in the situation. In Prolog, a choice must be made between a recursive loop or an iterative loop. In Prolog, iterative loops are modeled by using backtracking to force the looping sequence.

An elegant looping structure can be created by using recursion. In these cases, a counter can be used as the terminating condition to the recursive procedure. Using counters, a condition can be set either for when the counter reaches zero or for when the counter reaches a predefined number. The program that follows shows how to count from 1 to 100 using recursion:

```
/* Recursive Counter */
Domains
    list = integer*

Predicates
    make_list ( list, integer )
```

```

Clauses
make_list([],100) :- !.           % terminating condition
make_list([Number_1|Tail], Number) :-
    Number_1 = Number + 1 ,
    write(Number_1), nl ,         % This process is done 100 times
    make_list(Tail, Number_1).

GOAL
makewindow(1,2,3," Make a List ",0,0,25,80) ,
make_list(List, 0) ,
readchar(_),
clearwindow ,
write(List), nl.

```

This short program does two things. First, it creates a counter that counts from 1 to 100. On top of this, the program creates a list of all of the numbers it has counted. The predicate `make_list()` is defined as tail recursive, so this format can be used with most looping problems without worrying about running out of memory. As an example of how this can be put to use, take a look at the predicate that counts the number of words in a string:

```

Predicates
word_count ( string, integer, integer )
punctuation ( string )

Clauses
word_count(String, Count_in, Count_out) :-
    fronttoken(String, Token, Rest) ,
    not( punctuation(Token) ), ! ,
    Count_In_1 = Count_in + 1 ,
    word_count(Rest, Count_in_1, Count_out).

word_count(String, Count_in, Count_out) :-
    fronttoken(String, _, Rest), ! ,
    word_count(Rest, Count_in, Count_out).

word_count(_,X,X).

punctuation("!"). punctuation(",").
punctuation("."). punctuation("?").
punctuation(";"). punctuation("\").
punctuation("("). punctuation(")").
punctuation("-"). punctuation("$").

```

This program relies on the fact that `fronttoken()` fails if it is given an empty string to tokenize. The procedure is written to be tail recursive, and works with a goal similar to the one shown here:

```
word_count("Input string.", 0, Count)
```

With this call to `word_count()`, the counter is initialized to zero and the word count of the input string is returned as the third

argument. Notice how the program attempts to deal with punctuation by skipping tokens that are not words. While this method works for most sentences, the counter is thrown off by hyphenated words, floating-point numbers, and punctuation marks not accounted for in the predicate `punctuation()`. For a good exercise in parsing, enhance `word_count()` so that it can handle these particular situations.

Iterative Counters Iterative loops in Prolog not only deal with counters, but also are used to form other basic loops. Iterative loops work by bouncing back to the top of a looping sequence being forced by a failing condition. Often called a *Repeat/Fail loop*, these structures usually require a `repeat()` predicate to be modeled for the creation of the loop.

Returning to the basics of backtracking, when a call fails, Prolog returns back to the last nondeterministic call made. To create a loop out of this process, all that is needed is a predicate that will always succeed when it is backtracked to. In another light, a predicate that can produce an infinite number of solutions is one that will always produce a positive result when backtracked to. The predicate `repeat()` does just this:

```
repeat.  
repeat :- repeat.
```

This predicate effectively creates a situation in which, no matter what happens, another solution can always be found when the call is redone. If `repeat()` is encountered through backtracking, forward processing will always resume once `repeat()` has been processed.

Because a loop is created between `repeat()` and a condition that fails, these loops are called *Repeat/Fail loops*. Notice, however, that if the condition that fails can never become true (such as the predicate `fail()`), an infinite loop will be created between `repeat()` and the failing condition; Prolog simply cannot exit such a loop. Because of this, these loops are usually written in the format of a *Repeat/Failing-Condition loop*, where the call that causes the fail is a condition that will succeed once the looping process has been completed. The following diagram shows how these loops are modeled

```

loop :-
    repeat ,
    <process to be repeated> ,
    terminating_condition = yes.

```

Here, the “process to be repeated” is performed and a test follows. If the terminating condition cannot be met, backtracking takes processing back to the top of the loop. The processing is repeated until the “terminating_condition” can successfully be met.

Here’s an example showing how repeat() can be used to create a menu selection routine. The process allows the user to choose a menu item, then process that item. This process is repeated until the user chooses to exit the menu routine.

Notice how the processing of menu items is taken care of with the process_choice() clauses. These clauses are written in the format of a basic case structure, as outlined in Chapter 7.

```

Domains
    choice = integer

Predicates
    repeat
    menu
    get_choice ( choice )
    process_choice ( choice )

Clauses
    repeat. repeat :- repeat.

    menu :-
        repeat ,
        makewindow(3,2,1," Menu ",5,5,10,20) ,
        write("Choose an item:\n\n" ,
            " (1) Read File\n" ,
            " (2) Write File\n" ,
            " (3) Delete File\n" ,
            " (4) Create File\n" ,
            " (5) Exit" ) ,
        cursor(0,16) ,
        get_choice(Choice) ,
        removewindow ,
        process_choice(Choice) ,
        Choice >= 5.

    get_choice(C1) :-
        readchar(C) ,
        C1 = C - '0' ,      % Convert from ASCII to integer value
        C1 <= 5, ! ,
        write(C1).
    get_choice(C) :-
        beep, get_choice(C).

```


Since some predicates are designed to be nondeterministic (such as `repeat()`), it helps to exempt these predicates from being flagged. The keyword `nondeterm` is used for this purpose; it is placed in front of nondeterministic predicate declarations. Since `repeat()` is nondeterministic by nature, it is common to see it declared in this manner:

Predicates

```
nondeterm repeat
```

Another use of the `nondeterm` keyword is to use it when declaring global predicates. By definition, all global predicates are deterministic. This means that by default, global predicates can return only one solution for any given call. To create a global predicate that is nondeterministic you must precede the global declaration with the keyword `nondeterm`, such as with the following declaration:

Global Predicates

```
nondeterm get__part__numbers(symbol, integer) - (i,o)
```

On the other hand, you may sometimes want to force a predicate to be deterministic. The opposite of `nondeterm` is the keyword `determ`. `Determ` forces a predicate to be deterministic, even though it is nondeterministic by nature. One place you can take advantage of this is to declare database facts to be deterministic.

By default, database predicates are nondeterministic, which allows them to produce multiple solutions through backtracking. Following is a small program that shows how deterministic database predicates function. Notice how you need to add the new database facts with `asserta()` to ensure that the fact is at the top of the database stack. Even though the database fact now becomes deterministic (which means that no backtracking points can be placed when the fact is called), all the facts asserted for that fact will stack up in the database. When you remove the database facts, you can access the facts one at a time, as is shown in the second part of the clause of `count()` where the facts are retracted. This type of database structure allows you to create a "first in, last out" (or FILO) style of database access. Here is an example:


```

Database
    determ counter ( integer )

Predicates
    count
    nondeterm repeat

Clauses
    count :-
        asserta( counter(0) ), fail.    % initialize counter
    count :-
        repeat ,
            counter(X) ,                % current counter value
            X1 = X + 1 ,                 % add 1 to it
            asserta( counter(X1) ) ,    % assert it as new value
            write(X1), nl ,              % write new value
            X1 > 99 ,                    % count to 100
        repeat ,
            retract( counter(Y) ) ,    % count down from 100
            write(Y), nl ,
            Y = 1 ,                      % test for end of loop
        write("Done!\n").

    repeat. repeat :- repeat.

GOAL
    makewindow(1,2,3,"Counter",0,0,25,9) ,
    count.

```

For/Next Loops In programming, you will often want to repeat a process a certain number of times. If the number of times the task needs to be repeated can be determined when you are writing the program, then you can *hard code* the terminating number right into the program code. However, sometimes the actual number of iterations to take place is gained by processing the program. In this case, you cannot hard code the stopping condition into the program since it is not known until the program runs how many times the loop is to be repeated. You must create a loop that terminates based on a value that is calculated in the course of processing.

The For/Next loop can handle such a situation. Since the terminating condition is checked against a value that is passed into the call, you can set the terminating value during the processing of the program. The For/Next loop works much like a Repeat/Fail loop; a predicate is built so that a new solution can be found when it is backtracked to. The only difference is that repeat() can never fail. The predicate for() will fail at the end and, in this way, terminate the looping sequence. The for() predicate is written like this:

```

Predicates
  for ( integer, integer, integer )

Clauses
  for(Num, _, Num).
  for(Start, Stop, Number_out) :-
    Start_1 = Start + 1 ,
    Start_1 <= Stop ,
    for(Start_1, Stop, Number_out).

```

The for() predicate can now be used to create a loop in this manner:

```

Predicates
  for ( integer, integer, integer )
  loop

Clauses
  for(Num, _, Num).
  for(Start, Stop, Number_out) :-
    Start_1 = Start + 1 ,
    Start_1 <= Stop ,
    for(Start_1, Stop, Number_out).

  loop :-
    for(1, 5, X) ,
    write(X), nl ,
    fail.
  loop.

GOAL
  makewindow(1,2,3," For Loop ",0,0,10,20) ,
  loop.

```

The loop in the listing above is, in effect, saying "For 1 to 5, write the value of X." The value of X will always be the current status of the loop counter. Sometimes it will be necessary to use the counter; at other times it won't. In any case, the predicate for() allows you to access the current counter value.

Another use of a For/Next loop is to create *nested loops*. Nested loops can be easily modeled in Prolog using the for() predicate. The sample program that follows shows how you can write a procedure that quickly asserts a set of facts into the database:

```

Database
  fact ( integer, integer )

Predicates
  for ( integer, integer, integer )
  next ( integer, integer )
  assert_facts
  write_facts

```

Clauses

```

for(Num, _, Num).
for(Start, Stop, Number_out) :-
    Start_1 = Start + 1 ,
    Start_1 <= Stop ,
    for(Start_1, Stop, Number_out).

next(X,X).

assert_facts :-
    Test_1 = 5 ,
    Test_2 = 3 ,
    for(1, Test_1, Arg_1) ,
        for(1, Test_2, Arg_2) ,
            assert( fact(Arg_1, Arg_2) ) ,
            next(Test_2, Arg_2) ,
        next(Test_1, Arg_1) ,
    write("Done!\n") ,
    readchar(_).

write_facts :-
    fact(X,Y) ,
    writef("X = %   Y = %\n",X,Y) ,
    fail ; true.

```

GOAL

```

makewindow(1,2,3," For/Next ",0,0,25,80)
assert_facts ,
write_facts.

```

Following this program through in the Trace mode shows how backtracking is used to force a looping situation. The predicate `next()` simply makes a comparison with the current counter value and the stopping condition. If the two are equal, then `next()` will succeed and the loop will fall through.

In the predicate `for()`, the number "1" is hard coded into the code, causing the loop to be incremented by one with each iteration. Because of this, the loop always counts up one at a time. By adding another argument to the predicate, it is possible to specify the amount you wish the counter to increment each time the loop is processed. The predicate `for()` now looks like this:

Predicates

```

for ( integer, integer, integer, integer )

```

Clauses

```

for(Num, _, Num, _).
for(Start, Stop, Number_out, Inc) :-
    Start_1 = Start + Inc ,
    Start_1 <= Stop ,
    for(Start_1, Stop, Number_out, Inc).

```

Negation: Using Cut/Fail

Sometimes, you will want to ensure that a given circumstance will always fail. An example is in comparing two values. If the two values are equal, you will want to force backtracking. But if they are not equal, you will want to continue processing the two values. Using negation, coding such a procedure becomes easy:

```
check_integers(X,X) :- !, fail.  
check_integers(X,Y) :- X < Y, ! , ...  
check_integers(_,_) :- ...
```

Combining the two predicates `cut()` and `fail()` guarantees that if the values are equal backtracking will start (even though there may be alternate solutions to the call). The `cut` predicate prevents the other clauses from being tried, while the `fail` predicate causes backtracking. In the procedure just given, negation is assured if `check_integers()` is supplied with two equal values. If the arguments are not equal, or if they cannot be made equal through unification, then Prolog will continue its processing.

Care should be taken not to accidentally create a *Cut/Fail* situation in the body of your programs. It is easy to overlook that a `cut` has been placed and create a situation in which it is impossible for a condition to succeed after a `cut` has been passed. This situation will prohibit alternative clauses from being tried, and you will be left wondering why your program keeps failing.

Complex Data Structures

In Prolog, several varieties of data types and data structures can be used in advanced programs. These data objects include recursive data structures (introduced in Chapter 6), lists of multiple types, and arguments that need to be declared as reference types.

Reference Arguments In Prolog, a call will usually return a value that can then be passed on to other predicate calls. Arguments

that return values are known as *output arguments*. Sometimes in processing it is not possible for an output argument to return a value. This may happen because the information needed to calculate a value is not known at the time of the call. In these cases, the output argument will still be free when the call returns. Later, when this unbound variable is then used as an input argument to another call, a problem can occur because the argument is not bound to a value.

Prolog gains much power from the ability to *reference* these types of arguments. Even though the value of an argument may not be calculated at the time a return is made, Prolog may still be able to determine its value in later processing. This passing of unknown values does not necessarily stop Prolog from continuing with the processing that needs to be done.

If an output argument cannot be made to return a value, Prolog must create a reference to the variable in question and hope to solve for its value at a later date. While this method is extremely powerful, the creation of reference arguments generates more code and therefore should only be used when absolutely necessary.

Turbo Prolog normally does not allow the arguments of predicates to be referenced. However, to support this powerful feature, it allows you to declare certain arguments as reference types. To declare an argument as a reference type, you must precede the domain declaration with the keyword *reference*. The declaration that follows declares the domain *ref_list* to be a reference list of symbols:

```
Domains
  ref_list = reference symbol*
```

When compiling programs, you will sometimes encounter this error message: "Warning: The variable is not bound in this clause. (F10=ok, Esc=abort.)." This means that you need to declare a reference domain. You should take a close look at your code and see if it is absolutely necessary for you to use a "reference" domain. In some instances it may be, but nine times out of ten you will find that you can optimize your code so that reference domains are not needed.

Complex Lists In Turbo Prolog it is not possible to declare a list

to be composed of more than one standard data type. Even so, the clever use of compound objects will enable you to create lists that can hold any type of argument. To start with, a list made of other lists can be declared like this:

```
Domains
  integerlist = integer*
  element = integerlist
  list_of_lists = element*
```

This data structure can be used in many situations and closely models the multiple arrays used by other languages. Even though this is useful, what is sometimes needed is a list that can hold more than one object type. Although the list just shown is made up of other lists, each individual element is still the same type of data. The trick is to create a list whose elements consist of several types of compound objects. This can be done by declaring a list like this:

```
Domains
  element = i(integer) ; r(real) ; s(symbol) ; t(string)
  list = element*
```

You now have a list that can hold the data types integer, string, real, and symbol. Taking this a little farther, you can create a recursive definition, so that the list may even contain other lists. For example:

```
Domains
  element = i(integer) ; r(real) ; s(symbol) ; t(string) ;
           c(char)      ; l(list)
  list = element*
```

In this example, a list can contain lists inside of other lists. Using this technique, a list can be made as complex as you wish; there is virtually no end to the types of lists you can create. The best thing to do, however, is to keep the list as simple as possible. Once a list is capable of holding several different functor types, you'll need several clauses to handle each different element type. Not only this, but creating and keeping track of these lists can quickly become confusing.

Memory Management

When writing programs in Turbo Prolog, there are three memory arrays you must be aware of: the heap, the stack, and the trail. Each one performs a specific function, so a different array is used depending on the task you are performing. In addition to these arrays, Turbo Prolog also sets up storage for the *code array*, which stores the code that is generated when your program is compiled.

In Turbo Prolog, a *trail array* is allocated to keep track of binding and unbinding reference variables. Since most of the programs you write will not take advantage of this feature, you can afford to keep this array set to the minimum size defined by the system. By default, the trail array is set to 10 paragraphs (a paragraph is equal to 16 bytes of memory).

The *stack array* is mostly used for transferring arguments during the processing of your program. For example, if you write a recursive procedure to create a list, the list will be stored on the stack as it is being created. Once the list is no longer needed as an argument, the stack can then be freed of the memory used to store the list. The stack is also used to store pointers to untried subgoals. You will get a stack overflow message if you create a recursive predicate that cannot be optimized using tail-recursion elimination.

Normally the programs you write will run under the default stack setting made by the system (600 paragraphs). But if you are writing a program that creates large data structures or has a large search space, you may need to increase the size of your stack array. The Turbo Prolog system will indicate that this is needed by informing you that the stack has run out of available space and that you must reconfigure the size of your stack array.

The *heap array* is the location where most of the processing in your program takes place. The heap array is allocated after your program allocates the trail and stack arrays. Since a Turbo Prolog program uses up all the available RAM in your system, the heap makes use of all the RAM that is left over after the other memory arrays have been allocated.

In most of the programs you write, the memory allocation is of little concern. You will find that the Turbo Prolog default settings can handle most medium-sized applications. However, as your programs get larger, you will become more familiar with the error messages associated with memory overflows. If you get a heap over-

flow message as you are compiling a program, it may indicate that you need to think about using modules to compile your program. Stack overflows usually mean that you need to increase the stack array for your program.

While a heap overflow can indicate that your program has become too large to compile as a single file, it can also indicate that you need to restructure some of the code in your program. To cut down on the use of the heap, it is recommended that you code Repeat/Fail loops instead of recursive loops. This is especially important if you have a program that runs for an extended period of time. A tail-recursive clause can be optimized, but optimization is not 100 percent. If you have a program that needs to run for hours on end, a recursive clause will eventually exhaust the heap array. In this case, you will need to reformat the loop using fail() to force the looping sequence, and thus save on heap usage.

Once a program begins to have memory problems, you will need to figure out where the overflow is occurring. The built-in predicate storage() can monitor the three memory arrays consisting of the stack, the heap, and the trail. Here is a section of code that allows you to monitor where your memory is being eaten up. The predicate memory() takes a symbol for an argument and reports the current stack and heap sizes in bytes. Since the trail array is usually not a concern, this value is not used in the predicate memory(). However, it is not difficult to add this value to the memory usage report.

```
Predicates
    memory ( symbol )

Clauses
    memory(Place) :-
        shiftwindow(Current) ,
        shiftwindow(10) ,
        storage(S,H,_ ) ,
        writef("Stack: %   Heap: %   %\n",S,H,Place) ,
        shiftwindow(Current).

GOAL
    % Create memory size window
    makewindow(10,4,4," Memory Sizes ",0,40,10,40) ,
    ...
```

With memory(), you will need to initialize the output window in the goal of your program. After this, just make a call to memory() to get the sizes of the stack and heap. The single symbol

argument used in `memory()` provides a way to indicate the location in your program from which the `memory()` predicate is being called. The program that follows shows how `memory()` can be put to use:

```
Domains
    list = symbol*

Predicates
    run
    memory ( symbol )
    make_list ( list )

Clauses
    run :-
        write("Enter the elements to make up a list.\n" ,
              " (Enter a blank line to terminate the list)\n\n" ) ,
        make_list(List) ,
        clearwindow ,
        memory(after_list) ,
        write(List) ,
        fail.
    run.

    make_list([Head|Tail]) :-
        readln(Head) ,
        Head <> "" , ! ,
        make_list(Tail).
    make_list([]) :-
        memory(tail).

    memory(Place) :-
        shiftwindow(Current) ,
        shiftwindow(10) ,
        storage(S,H,_ ) ,
        writef("Stack: % Heap: % %\n",S,H,Place) ,
        readchar(_ ) ,
        shiftwindow(Current).

GOAL
    makewindow(10,4,4," Memory Sizes ",0,40,10,40) ,
    makewindow(1,2,3," Make A List ",0,0,25,80) ,
    memory(beginning) ,
    run ,
    memory(end).
```

Options Menu The Turbo Prolog system sets up basic sizes for the different memory arrays. However, the system provides both menu choices and compiler directives to allow you to customize these memory settings.

The heap, the trail, and the code arrays can all be customized from within the source code of your program by using the associated compiler directives. All three directives take an integer value as an argument (given in paragraph units), and each sets the named

array to the desired size. The following directive placed at the top of your program code overrides the default setting and sets the code array to 2,000 paragraphs:

```
code = 2000
```

A more convenient way to set memory array sizes is to use the Options menu. As submenu to the Options main menu choice, the Compiler Directives menu allows several directives to be set interactively. Choosing the next submenu, Memory allocation, enables you to customize the sizes of the code array, the stack array, the heap array, and the trail array in your program. While these menu options are a handy way to customize memory allocations, any directives placed in the code override the settings made in these menus.

Debugging Projects

When a program becomes too big to compile as a single file, you are forced to break up your program into separate modules and compile the program as a project. While modular programming has many advantages, one disadvantage is that you can no longer trace a project once it is compiled; tracing .EXE files is not possible in Prolog.

There are two different approaches to making sure that large programs run correctly. The first approach is to debug each module as you write it. The second is to debug modules that have been compiled as part of the final, executable program.

Debugging Modules When you create a project, you will code one module at a time, then link all the final modules together to form your final product. The most reliable way to ensure that your program is up to par is to be sure that each module operates correctly before you compile the project. This can be done by debugging each module while you are coding it.

Each program module will contain various predicates made up of both facts and rules. In a program module, the rules may contain

calls to predicates that are defined in other modules. This is known as making a *global call*. Most likely, the purpose of a call (either local or global) is to obtain information from the predicate being called. In projects, once a call is made to a different program module, you lose the ability to trace the actions of the file you are working on.

The trick to debugging a module is to simulate the returns that the global calls make. In other words, each global call will come back with some information, which you will then use in processing the program. If you can simulate the values that the global calls return, there is no need to make a global call. You can trace the program locally and be sure the module works before you attempt to compile it and link it in the project.

It is not difficult to simulate the global calls that are made inside of a module. All that you need to do is add a fact or set of facts to the end of the program to represent the values that are returned by the global calls. For example, suppose you have a call to a global predicate that is supposed to calculate a customer's discount. The call might look something like this:

```
...
get_discount_rate(Customer, Id, Rate),
...
```

Before this call is made, you will know the customer's name and identification number. To model this global call, you can add this fact to the end of the module:

```
get_discount_rate(—, —, 0.23).
```

Now your program doesn't need to make a global call, and you can easily trace and debug each module as you go along.

It's a good idea to place all of your simulation predicates in one group at the end of each module. Then, when you are finished with debugging, you can just "comment out" (place comment markers around) the added predicates. Since Prolog ignores everything within comment markers, the extra code will not confuse the compiler. Also, if you need to come back and fix a module, all your simulated predicates will still be there for you to use and inspect.

Debugging Executable Programs It is not always easy to trace a problem by simulating global calls. Sometimes your entire program will compile and run fine, except that one section keeps coming up with the wrong results. In this case, you will not want to trace and debug each individual module again; you will just want to trace the section that is giving you the problem.

Much like simulating global predicates, you can simulate a Trace window in programs that are compiled. The idea is to write out the bindings of certain variables as you go along, making sure that the values are correct. This can be done by creating a Trace window in the Goals section of your program. Then, in strategic places in the program, write the variable bindings to this window. Here is the code that can be used to accomplish this task:

```
Global Predicates
  debug_prj ( string ) - (i)

Clauses
  debug_prj(Str) :-
    shiftwindow(Current) ,      % get current window
    shiftwindow(13) ,           % go to debug window
    write(Str), nl ,            % write out string
    readchar(_) ,               % wait for input
    shiftwindow(Current).        % go back to current window

GOAL
  makewindow(13,7,7," Debug ",0,40,10,40) ,
  ...
```

Using this method, you can create a Debug window that can be referred to from anywhere inside the program. The window should be placed in the upper-right corner of the screen or away from the program's screen activity. Now all that needs to be done is to call `debug—prj()` with a string to be displayed and you're on your way to debugging an executable program.

NOTE: The easiest way to get the proper string into the `debug—prj()` predicate is to use the built-in predicate `format()`, which will create the string you want to print out. In the places where you want to print out debugging information, you will first call `format()`, then `debug—prj()`. Here's an example:

```
run :-
  get_customers_rate( p(First, Last, Rate) ) ,
```



```
% debugging code
format(Str,"First: % Last: % Rate: %", First, Last, Rate)
debug_prj(Str) ,
...
```

13 *Using the BGI and the Toolbox*

With all of Turbo Prolog's syntax under your belt, you now have the knowledge needed to write applications in Turbo Prolog. As your programming skills become more proficient, you will no doubt want to add some dazzle to your final products. Perhaps you will want to make a fine user interface for a program or generate some bar or pie charts for a report.

With Turbo Prolog, adding these finishing touches is easy, because the tools to do so are given to you. With the addition of the Turbo Prolog Toolbox, sophisticated menus and data entry screens are easily included into your programming applications. Adding graphics to your programs is easy because the tools to create graphic images are included with the Turbo Prolog compiler.

Using the Turbo Prolog Toolbox and the Borland Graphics Interface (referred to as the BGI) is fairly similar. Both are tools that you add to your programs, and both are used by calling the appropriate predicates to perform the function that you want to carry out.

The BGI

The Borland Graphics Interface is a library of graphics tools made up of over 70 predicates. There is nothing extra to buy to take advantage of these tools—they are built into the Turbo Prolog compiler. Adding graphics to your program is handled entirely by making calls to the predicates in the BGI library. However, before you can write programs that use graphics, you must first have a computer system that is capable of displaying graphic images. Since most personal computer systems today provide graphics support, it is likely that your system is capable of using the BGI. Before continuing with this section, take the time to find out if your system has graphics capabilities, and whether your system supports monochrome, CGA, or EGA graphics.

Text Versus Graphics

On a computer screen, there is a big difference between displaying text and drawing graphics. Normally, output written to the screen is done in *text mode* with text characters. In contrast, the graphics mode is used to draw special images that cannot be achieved with the characters your system uses to write text.

Text Mode In text mode, the visual display is broken up into separate *cells* that are divided among the rows and columns of your computer screen. A normal text display makes use of 25 rows and 80 columns, creating a total of 2,000 cells for the positioning of text on your screen ($25 \times 80 = 2,000$).

By using special video adapters, different text modes can be achieved. Your screen can be divided into more rows and columns, giving modes such as 43×80 (43 rows by 80 columns), 50×80 , and so on. In text mode, you have the ability to move the cursor to any one of the cells provided by a particular mode, such as the 2,000 positions provided from the usual 25×80 mode. You are also capable of outputting any printable character to any of the cells in the display.

Graphics Mode In graphics mode there is no cursor to keep the screen position like there is in text mode. Instead, a screen pointer known as the *current position* (the *CP*) keeps track of the screen location. This screen pointer can be moved about the display, and graphic images can be drawn in specific locations.

In graphics mode, your screen is broken up into thousands of points, known as *pixels*. The number of pixels on your screen is determined by the *screen resolution* of your graphics system. The higher the number of pixels, the higher the resolution of your system and the more crisp your graphics will be. Each pixel is capable of being turned either on or off, and with color systems, pixels can display specific colors. Graphic images are achieved by turning on, and perhaps coloring, the appropriate pixels to make a picture.

Graphics Cards

Several types of graphics adapters exist for IBM personal computers and compatibles. The most popular graphics cards are the Color Graphics Adapter (known as a CGA card) and the Enhanced Graphics Adapter (known as an EGA).

Most personal computer systems make use of one of these two graphics cards and the BGI supports both. The discussion on how to initialize and draw graphics is based on these two adapters. If your system happens to make use of a different graphics adapter, the material in this chapter still applies, but you may need to refer to the *Turbo Prolog User's Guide* for details concerning the correct driver to use with your graphics system. All the following graphics adapters are supported by the BGI:

- Color Graphics Adapter (CGA)
- Enhanced Graphics Adapter (EGA)
- Multi Color Graphics Adapter (MCGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter (Monochrome Graphics)
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

CGA Graphics The CGA card was the first graphics adapter introduced for use with the IBM personal computer. This video card supports a single text mode (25×80) and two separate graphics modes. When you place a CGA system into graphics mode, you can specify either high-resolution or low-resolution graphics.

When *low-resolution mode* is enabled, your screen is broken up into 320 pixels across and 200 pixels down, giving a total screen resolution of 320×200. When a screen is initialized into low resolution on a CGA system, you can display four colors at a given time, one of which is for background. (For more on color, see “Colors and Graphics” later in this chapter.)

In *high-resolution mode* CGA, your screen is split up into 640 pixels × 200 pixels (a resolution of 640×200), and you have the ability to display a single color on a black background.

EGA Graphics With an EGA system, your graphics capabilities are greatly expanded. Like the CGA system, an EGA system can be initialized into one of two graphics modes. Low-resolution graphics on an EGA system divides your screen into 640×200 pixels, while high-resolution graphics gives you a resolution of 640×350. Both high- and low-resolution graphics of an EGA system allow 16 separate colors to be displayed.

The major difference between the two graphics modes on an EGA card is the number of buffered pages provided. In low resolution, four complete pages of graphics screens can be stored in your EGA card's memory. This allows you to create graphic images on three different screens while displaying a fourth. These three extra pages give you the power to draw images on a hidden screen, then quickly change from one graphics screen to another without having to wait for the image to be drawn. In EGA high resolution mode, two graphics pages are available, one hidden and one displayed.

Initializing Graphics Mode

In Turbo Prolog, when you want to change from a text screen to a graphics screen, you must physically initialize the graphics mode you wish to use. With the BGI, the graphics mode is initialized by

making a call to the built-in predicate

```
initgraph()
```

This predicate makes use of five arguments and sets up the graphics mode by way of a *graphics driver*. In addition to providing a particular graphics mode, you must also supply `initgraph()` with the proper graphics driver name.

Graphics Drivers To initialize a graphics screen, the BGI must use a specific graphics driver to work with your particular graphics system. The graphics drivers are the Turbo Prolog files that have a BGI file extension. The BGI uses six drivers to support the different graphics adapters:

```
ATT.BGI  
CGA.BGI  
EGAVGA.BGI  
HERC.BGI  
IBM8514.BGI  
PC3270.BGI
```

In a call to `initgraph()`, the driver argument tells the BGI what type of graphics card to expect and the mode argument specifies the mode you have chosen. The drivers to be used with CGA and EGA systems are `CGA.BGI` and `EGAVGA.BGI`, respectively. Each driver supports a number of modes, as shown in Table 13-1.

GRAPDECL.PRO To help clarify your program code, Borland has included a declarations file to be used with the BGI. The file `GRAPDECL.PRO` (short for graphics declarations) consists solely of a Constants section. Its use makes it possible to refer to the different values used in BGI predicates by a symbolic name rather than by a cryptic set of numbers. For example, rather than using the number 1 to indicate EGA high-resolution graphics (the number required by `initgraph()`), you can use the mnemonic constant `egaHI` that is declared in `GRAPDECL.PRO`. To take advantage of these graphics constants, include the `GRAPDECL.PRO` file in the programs that use graphics.

In Table 13-1, the columns "Driver Name" and "Mode Name" represent the constant declarations made in GRAPDECL.PRO.

NOTE As previously stated, the BGI supports several graphics cards other than CGA and EGA adapters. However, to keep the examples clear, only the CGA and EGA systems are shown here.

Calling Initgraph() The BGI predicate `initgraph()` makes use of five arguments. The first argument to `initgraph()` supplies the graphics driver value, while the second argument dictates the mode that the graphic image will be drawn in. The third and fourth arguments to `initgraph()` are output arguments; they return the driver and mode when `initgraph()` is used to automatically detect the graphics drive and mode. The fifth argument to `initgraph()` points to the DOS directory where the BGI driver resides. On a hard disk system, this argument should be a drive letter followed by a DOS path. On a floppy disk system, this argument should be the disk drive where the graphics drivers are located.

To enter a graphics mode, all you need to do is to call `initgraph()` with a valid BGI driver name, a mode that can be associated with that driver, and a path pointing to where the driver can

Driver File	Driver Value	Driver Name	Modes	Mode Name	Resolution
CGA.BGI1	1	cga	0	cgaC0	320×200
			1	cgaC1	320×200
			2	cgaC2	320×200
			3	cgaC3	320×200
			4	cgaHI	640×200
EGAVAG.BGI	3	ega	0	egaLO	640×200
			1	egaHI	640×350

Table 13-1. CGA and EGA Graphics Modes

be found. Here's an example of calling `initgraph()` to initialize an EGA system using high-resolution graphics:

```
initgraph(3,1,_,_, "c:\\prolog\\bgi")
```

If you have included `GRAPDECL.PRO` in your program, the call to `initgraph()` can be made more readable by using the constant declarations made in that file:

```
include "c:\\prolog\\bgi\\grapdecl.pro"

Constants
    bgi_drivers = "c:\\prolog\\bgi"

GOAL
    initgraph(ega,egaHI,_,_,bgi_drivers) ,
    ...
```

As you can see, making your own constant declarations to lead to the BGI drivers can also make your program more transparent. Once you step into graphics mode, you are free to call on any of the BGI predicates to draw lines, circles, create bar graphs ... the list is only limited by your imagination.

Using Detectgraph() Calling `initgraph()` is easy if you know in advance the type of graphics system your program is going to run on. However, if you're developing an application for general distribution, knowing all the systems that your program could be used on is a little tougher. In such a case, it's not possible to hard code a particular driver or supply a specific graphics mode to produce the best results. Because of this, the BGI provides a built-in predicate for just this situation.

```
detectgraph()
```

The BGI predicate `detectgraph()` makes a check with the hardware in the computer and returns values according to the video system detected. Taking two output arguments, `detectgraph()` returns the value of the driver needed to draw graphics and a mode number corresponding to the highest resolution for the specific card detected.

With `detectgraph()`, initializing a graphics mode can be generalized to work with any graphics system that the BGI supports. The initialization of a graphics mode on any system can look like this:

```
include "c:\\prolog\\bgi\\grapdecl.pro"

Constants
    bgi_drivers = "c:\\prolog\\bgi"

GOAL
    dectectGraph(Driver, Mode) ,
    initgraph(Driver, Mode,_,_,bgi_drivers) ,
    ...
```

Normally, you will wish to draw graphics in the highest screen resolution available. The highest resolution of a system always creates the most crisp image that the system is capable of drawing. It's a simple rule: the higher the resolution, the more crisp the graphics.

Since `detectgraph()` returns a value that corresponds to the highest mode available, things work out quite nicely. However, in certain circumstances—such as with CGA systems, where the high resolution only supports two colors—you may want to override the mode recommended by `detectgraph()`. However, this is no problem. All you need to do is set up a procedure that adjusts the graphics mode according to the driver value returned. Your graphics initialization code now looks like this:

```
include "c:\\prolog\\bgi\\grapdecl.pro"

Constants
    bgi_drivers = "c:\\prolog\\bgi"

Predicates
    set_mode ( integer, integer, integer )

Clauses
    set_mode(1,_,0) :- !.    % CGA system
    set_mode(_,X,X).        % all others

GOAL
    dectectGraph(Driver, Model) ,
    set_mode(Driver, Model, Mode) ,
    initgraph(Driver, Mode,_,_,bgi_drivers) ,
    ...
```

Once you step into graphics mode, you can draw and color your screen in almost any fashion you can think of. When you are

finished drawing, you can go back to text mode and continue with what you were doing before you entered the graphics system. To get back to text mode, issue the built-in predicate to terminate the graphics session:

```
closegraph()
```

When `closegraph()` is called, the graphics screen is put away and you are returned to the text mode you were using before `initgraph()` was called.

Viewports

When a graphics mode is initialized, the entire screen is cleared and set to a graphics screen. The cursor position is changed into a screen pointer and is *homed* in the default *viewport*. Much like the windowing system of text mode, viewports are used to select specific sections of your screen in which to draw graphics. When graphics are initialized, the default viewport is set to the entire screen.

Once in the viewport, the “y” coordinate increases as you move down the screen, and the “x” coordinate increases as you move to the right on the screen. The number of units on each axis is identical to the number of pixels on that axis. When you step into graphics mode, or create a new viewport, the screen pointer (CP) is always placed in the upper-lefthand corner of that viewport, which is position 0,0.

While viewports and text windows have their similarities, they also have their differences. Viewports are not buffered, meaning that if you create one viewport on top of another, the information in the viewport that is written over will be lost. Another difference between windows and viewports is that window coordinates are specified in terms of text cells. Viewport coordinates are given in terms of *X,Y coordinates*, and these values can be directly translated into the pixel positions on the screen.

To define a viewport other than the default one set up by `initgraphic()`, you must make a call to `setviewport()`. The BGI predicate `setviewport()` takes five arguments, defining the size of the

viewport to be made. Using `setviewport()` is similar to setting up a window size with a call to `makewindow()`. However, all sizes are given in pixels and care must be taken not to go outside the boundary of your graphics screen when creating the new viewport. A call to `setviewport()` takes the form of

```
setviewport(left, top, right, bottom, clip_flag)
```

A call to `setviewport()` creates a rectangular viewport whose upper-lefthand corner is defined by the coordinate (left, top) and lower-righthand corner by the coordinate (right, bottom). In the process of creating a new viewport, the area is cleared where the viewport is placed and the current position is set to 0,0 (the upper-lefthand corner).

The fifth argument in `setviewport()` specifies whether the images should be cut at the edges or not. The argument `clip_flag` takes an integer and turns the *clipping* in the viewport either to on or off. If the argument is set to zero, then images drawn in the viewport will be clipped at the edges. This means that if an image drawn into the viewport exceeds the edges of the viewport, the image will be truncated at the edge of the viewport. A nonzero argument turns the clipping ability off, meaning that all parts of the images will be drawn, regardless of whether the viewport boundaries are exceeded or not.

Colors and Graphics

By now, you should feel comfortable specifying the text and window color attributes in your programs. Colors used with graphics are specified in much the same way, but the method for calculating the actual color varies from one graphics driver to the next. There are basically two approaches used to calculate colors produced by graphics output. The first method—the CGA method—is used by systems that incorporate CGA cards, AT&T graphics adapters, MCGA cards, and the cards used inside IBM 3270 personal computers. Slightly different from the CGA coloring method, the EGA method applies to only EGA and VGA graphics systems.

CGA Coloring Scheme A CGA card can be placed into one of two different graphics modes. In low-resolution graphics (320×200 pixels), the adapter is capable of producing four colors. While one of the colors must specify the background color, the other three colors may be used to draw with.

The background color can be set to any one of the 16 different colors shown in Table 13-2. Setting the background color on a CGA system is done with the BGI predicate

```
setbkcolor()
```

This predicate takes an integer value and sets the background of the screen to the corresponding color found in Table 13-2. If you wish, you may substitute the constant name shown for the integer value if you have included `GRAPDECL.PRO` in your program.

The remaining three colors on a CGA graphics display are set by choosing one of the predefined color palettes associated with

CGA Color	Color Value
Black	0
Blue	1
Green	2 ✓
Cyan	3 ✓
Red	4 ✓
Magenta	5 ✓
Brown	6
Lightgrey	7
Darkgrey	8
Lightblue	9
Lightgreen	10
Lightcyan	11
Lightred	12
Lightmagenta	13
Yellow	14 ✓
White	15

Table 13-2. CGA Colors

CGA cards. Choosing the palette is done by supplying `initgraph()` with the chosen CGA mode value or equivalent `GRAPDECL.PRO` constant. Somewhat limiting in color selection, each of the four CGA palettes has a set color pattern. The palettes and their color schemes are shown in Table 13-3.

The high-resolution mode of the CGA card makes use of 640×200 pixel resolution. When a CGA card is initialized to high resolution, only two colors are made available: the foreground color and the background color. The background color is automatically set to black and the foreground color is set to any one of the 16 colors shown in Table 13-2. Oddly enough, the foreground color in CGA high resolution is set with a call to the BGI predicate

`setbkcolor()`

Taking a single integer value (or `GRAPDECL.PRO` constant), the foreground drawing color in CGA high resolution can be changed using `setbkcolor()`.

EGA Coloring Scheme Enhanced Graphic Adapters (EGAs) live up to their name by greatly enhancing the graphics possibilities over their CGA counterparts. To start with, the high-resolution mode on an EGA display is a crisp 640×350. Not only this, but you are free to create your own color palette (consisting of 16 colors) from a selection of 64 different colors.

Driver Mode	Palette Number	Palette Color Values		
		1	2	3
CGAC0	0	Lightgreen	Lightred	Yellow
CGAC1	1	Lightcyan	Lightmagenta	White
CGAC2	2	Green	Red	Brown
CGAC3	3	Cyan	Magenta	Lightgrey

Table 13-3. CGA Color Palettes (Low-Resolution Mode)

When you initialize a graphics screen on an EGA system, the default palette shown in Table 13-4 is used to draw colors. The palette colors are kept in a list, which is declared to be from the domain "bgi_iList". Turbo Prolog automatically creates the domain declaration for palette lists as

```
bgi_iList = integer*
```

On an EGA system, the palette consists of a list of 16 integers. The first element in the list dictates the background color, while the other 15 elements declare the colors that can be used to draw images with. Selecting a drawing color is accomplished by choosing an entry in the palette list, ranging from 1 to 15. When a color is selected, the corresponding element in the list is looked up, and the color is set according to the value of that list element.

EGA Color	Default Palette Values	Color Value
ega_BLACK	0	0
ega_BLUE	1	1
ega_GREEN	2	2
ega_CYAN	3	3
ega_RED	4	4
ega_MAGENTA	5	5
ega_BROWN	6	20
ega_LIGHTGREY	7	7
ega_DARKGREY	8	56
ega_LIGHTBLUE	9	57
ega_LIGHTGREEN	10	58
ega_LIGHTCYAN	11	59
ega_LIGHTRED	12	60
ega_LIGHTMAGENTA	13	61
ega_YELLOW	14	62
ega_WHITE	15	63

Table 13-4. EGA Colors

Selecting a Drawing Color With either low-resolution CGA modes or EGA systems, setting the background color is achieved by calling the predicate

```
setbkcolor()
```

This BGI predicate takes a single integer value (otherwise known as GRAPDECL.PRO-defined constant) and sets the background color accordingly. To inspect the current background color value, call the BGI predicate

```
getbkcolor()
```

with a single free variable.

The following predicates from the BGI allow you to inspect the color palette and change the EGA color palette. Predicates are also supplied to change a specific palette entry or change the current palette entirely.

```
getpalette()  
setpalette()  
setallpalette()
```

Choosing a specific drawing color is done by calling the predicate

```
setcolor()
```

This predicate works with all graphics adapters, and takes a single integer argument corresponding to one of the colors in the palette table.

With a CGA card in low-resolution mode, `setcolor()` can be called with a value ranging from 0 to 3, with 0 setting the drawing color equal to the background color. The drawing color is set according to the corresponding color in the current CGA palette. The CGA palette is chosen by the mode with which `initgraph()` is called (see Tables 13-1 and 13-2).

In EGA systems, `setcolor()` has a workable range of 0 to 15, again with zero defining the background color. When a color is chosen on EGA systems, the corresponding element in the palette list is looked up and the drawing color is set to the value of that element.

Elements in the EGA palette are simply indexes to the actual color to be chosen. Choosing a color value of 14 does not set the color to yellow (the color of the value 14), but rather sets it to the value that is located in the fourteenth location of the palette list. Color values on an EGA system range from zero (black) to 63 (high-intensity white).

To inspect the current color selection, use the predicate

`getcolor()`

Calling `getcolor()` with a single free variable returns the integer value of the current color being used.

Drawing in Graphics Mode

Once you initialize the graphics screen and choose a color to draw with, you're set to begin creating graphic images. To aid in your creations, the BGI provides over 20 predicates to help draw different graphics shapes. In addition to creating graphics images, the BGI also provides several powerful predicates that can be used to set up pie charts, draw bar graphs, and even create reports with 3-D bars. You have the ability to select the lines you draw, and you can fill the images on the screen with system- or user-defined patterns.

After the graphics screen has been initialized and you have chosen a drawing color, you're ready to start creating graphic images. Since showing all the BGI drawing predicates and their variations is beyond the scope of this chapter, a brief introduction to the line-drawing features is given. After this, you can begin to experiment with the BGI features and predicates on your own.

Lines The BGI provides three separate line-drawing predicates for creating straight lines. Two of these predicates,

`linere()`

and

`lineto()`

make use of the current screen position to draw a line. The predicate `linere()` takes two integer arguments: an x distance and a y distance. Starting at the current position (CP), a line is drawn to a point calculated by the given distances. The predicate `lineto()` simply draws a line to the given x,y coordinate, starting at the CP.

The third line-drawing predicate

`line()`

takes four arguments. The first two indicate a starting x,y coordinate and the second two represent a finishing x,y coordinate.

Lines drawn by the BGI can be set to variable sizes, patterns, and styles. Setting these three attributes is taken care of with the BGI predicate

`setlinestyle()`

The first argument of `setlinestyle()` takes an integer value corresponding to the chosen line style. Table 13-5 defines the built-in line styles. The second argument to `setlinestyle()` is a 16-bit value that sets the pattern of the line. Using a hex value, the “*Upattern*” can be set to \$FFFF if you want a solid line pattern. The last argument in `setlinestyle()` corresponds to the thickness you want for your line. Two constants are defined in `GRAPDECL.PRO` for setting the line thickness:

`norm_WIDTH = 1-pixel-wide lines`

`thick_WIDTH = 3-pixel-wide lines`

In addition to drawing lines, the BGI gives you the power to draw multi-sided shapes and images. From arcs, to circles, to

GRAPDECL Name	Value	Description
<code>solid__LINE</code>	0	Solid line
<code>dotted__LINE</code>	1	Dotted line
<code>center__LINE</code>	2	Centered line
<code>dashed__LINE</code>	3	Dashed line
<code>userbit__LINE</code>	4	User-defined line

Table 13-5. *Line Styles*

polygons, the BGI can draw just about any image you can think up. Also, all the shapes you draw with the BGI may be filled with a pattern of your choice. The fill can either be on the inside or on the outside of the shapes you draw. Turbo Prolog provides five system-defined patterns to be used with the BGI. In addition to these useful patterns, you are free to design your own fill patterns to be used with any of the graphics applications you design. Take your time to browse through the chapter on BGI in the *Turbo Prolog User's Guide* for more information on what the drawing and painting features provide.

When drawing graphics, the BGI allocates a block of memory to be used for graphics. Most of the time, this allocated space is more than enough for your graphics needs. However, sometimes you will find that your graphic images demand more space than Turbo Prolog provides by default. When this happens, memory overflows can occur, leaving your program listless. The BGI provides a special predicate to help you out in such a situation:

```
setgraphbufsize()
```

The predicate `setgraphbufsize()` takes an integer for an argument and adjusts the graphics buffer size accordingly.

Writing Text in Graphics Mode

Besides drawing graphic images with the BGI, you can also display written text on a graphics screen. Unless specified, the output text

characters come from the default *bit-mapped* character set. Also available for writing are the *stroked fonts*, which can produce a more elegant version of the text you are writing.

Bit-Mapped Fonts Versus Stroked Fonts The two types of fonts supplied with the BGI are vastly different. While the bit-mapped font works for most applications, customized text is best written with stroked font characters.

The bit-mapped font is part of the Turbo Prolog library, so there is no need to include any extra files to output these characters. The one drawback to using the bit-mapped font is that the characters are drawn via a matrix that is mapped out in pixels. While the characters look good when they are written as small text, enlarging them reveals their blocky design. By default, the bit-mapped characters are mapped into an 8×8 pixel region.

Stroked font characters are a different story. Since the fonts are made up of a series of vertices, the characters do not lose their shape when they are enlarged. Not only this, the stroked fonts come in a variety of different styles, allowing you to create individualized text output.

Stroked fonts are kept in separate BGI files. Turbo Prolog files ending with a CHR extension are the stroked-font files, and consist of the following:

GOTH.CHR
LITT.CHR
SANS.CHR
TRIP.CHR

Unlike bit-mapped fonts, stroked fonts must be read from a file at runtime or linked during compilation. The section called “Adding the BGI to Your Programs” has more on this subject.

Selecting the font, the text size, and the direction of the text is accomplished with the predicate

```
settextstyle()
```

This BGI predicate uses three integer arguments and sets the character font according to the first argument. Table 13-6 describes the

GRAPDECL Name	File	Value	Description
default_FONT	(None)	0	8×8 bit-mapped font
gothic_FONT	GOTH.CHR	1	Stroked gothic font
sansserif_FONT	SANS.CHR	2	Stroked sans-serif font
small_FONT	LITT.CHR	3	Stroked small font
triplex_FONT	TRIP.CHR	4	Stroked triplex font

Table 13-6. *Graphics Fonts*

allowable input and shows the complementary GRAPDECL.PRO constant declarations.

The second argument to `settextstyle()` dictates the direction in which the text is written to screen. A zero indicates that the text is to be written out horizontally, and a 1 indicates that the text is to be written out vertically.

The last argument in `settextstyle()` indicates the size of the character to be written. This argument affects bit-mapped fonts differently than it does stroked fonts. With bit-mapped fonts, a zero or 1 in the third argument position indicates that the characters should be written in the default size of an 8×8 pixel box. If a 2 is placed in this position, the characters double in size to create a 16×16 pixel character matrix. A 3 in this position triples the size, and so on until 10 is reached. The maximum size for a bit-mapped font is ten times the default size.

The stroked-font size is handled differently than the bit-mapped fonts. With stroked fonts, the size is adjusted according to a *character magnification specification*. By default, this magnification is 4. When using stroked fonts, the default magnification can be achieved by using a zero in the size argument. A custom character magnification setting can be set by calling the predicate

`setusercharsize()`

This predicate uses four arguments and sets the character magnification accordingly.

Writing Text The regular output predicates `write()` and `writeln()` work for most graphics cards. However, some graphics adapters do not support their use when outputting graphics text. To overcome this limitation, the BGI provides two predicates to be used when outputting text in graphics mode:

`outtext()`

and

`outtextxy()`

The predicate `outtext()` places the text string at the current screen position, while `outtextxy()` allows you to specify a specific x,y coordinate. If you want to format the text before printing it out, use the built-in predicate `format()` to adjust the text to your liking. After this, you can supply the formatted string to `outtext()` or `outtextxy()` for printing.

Adding the BGI to Your Programs

After you have written and tested the graphics additions made to a program, you must decide how you're going to compile the BGI into your final program. Since both the graphics drivers and the stroked character fonts are found in special files, special steps must be taken to include the BGI procedures into your executable programs.

Compiling the Text Fonts If you are writing text in your graphics applications, you should know in advance which (if any) stroked character fonts are used in the program. If you are only using the bit-mapped character font, then there is nothing special to include at link time. If, however, you're using the special stroked character fonts, you'll have to decide whether to load the fonts in at runtime or whether you want to link them into your executable program.

If you want to include the stroked character fonts at runtime, `settextstyle()` will allocate the appropriate amount of memory for

the font when it is loaded in from the specified CHR file. The disadvantage to this procedure is that all the character fonts used in your program must be supplied with the final program. The advantage is that your executable file will be smaller without the fonts compiled into it.

An easier way to include the stroked fonts into your program is to compile the needed fonts directly into your final program. Including the fonts in your executable program will increase the final program size, but will also allow you to have a file that does not need to include special character font files at runtime. To include the stroked fonts into your executable program, you must put the compiler directive “bgifont” at the top of your program, and you must specify the fonts “*public name*” is to be linked in (see the section on “BGI Object Files” for more information). Each stroked font used in your program will have a separate bgifont directive, for example:

```
bgifont "c:\prolog\bgi\__small__font__far"
```

Compiling the Graphics Drivers Including the graphics drivers into your executable programs is similar to including the character fonts, except that you can only compile a single driver into a final executable program. If you want your final program to be distributed to several different people (and thus, perhaps, several different system configurations), the only way to be sure that the program will run on all systems is to load the correct BGI driver file in at runtime. Choosing the correct driver is done with a call to `initgraph()`, as described earlier in this chapter.

If you want to compile a single driver into the executable program, you must use the compiler directive `bgidriver` to load in the correct driver at compiletime. The directive `bgidriver` is given a BGI driver public name, and takes the form of the example given here:

```
bgidriver "c:\prolog\bgi\__EGAVGA__driver__far"
```

BGI Object Files To link the stroked character fonts and BGI graphics drivers into an executable program the fonts and drivers must first be in an object file format. All the object files for the BGI are supplied in the file `BGI.LIB` (located on your Turbo Prolog

disks), but they must first be extracted before they can be used.

Included with Turbo Prolog is a utility that extracts the OBJ files from the BGI.LIB file. BGIOBJ.EXE is the utility used to complete this job. If, for example, you want to include the small_FONT in your compiled programs, you must extract the small_FONT OBJ file with the following command:

```
BGIOBJ /F litt
```

Giving this DOS command generates a file called LITTF.OBJ, and it is given the public name of _small_font_far. The public name can now be supplied to the compiler directive bgifont, and you're ready to compile and link the font into your final program. Table 13-7 lists the BGI drivers and fonts by both their GRAPDECL constant names and associated public names, as well as the object file names that they extract to.

Turbo Prolog Toolbox

Like the BGI, the Turbo Prolog Toolbox is a set of *tools* that can be put to use in your Turbo Prolog programs. Unlike the BGI, the Toolbox comes as a separate package that must be purchased separately from the Turbo Prolog Compiler.

GRAPDECL Name	Public Name	Object File Name
default_FONT	(None)	(None)
gothic_FONT	_gothic_font_far	GOTHE.OBJ
sansserif_FONT	_sansserif_font_far	SANSF.OBJ
small_FONT	_small_font_far	LITTF.OBJ
triplex_FONT	_triplex_font_far	TRIPF.OBJ
cga	_CGA_driver_far	CGAF.OBJ
ega	_EGAVGA_driver_far	EGAVGAF.OBJ

Table 13-7. Graphics Drivers

Another, more pertinent difference between the Toolbox and the BGI is that the Toolbox is written almost entirely in Turbo Prolog. This means that many of the tools can be customized to suit your needs. Also, you are free to inspect the code to see how certain procedures are written by experienced Turbo Prolog programmers.

The Turbo Prolog Toolbox is broken up into six different sections. Tools are provided for creating user interfaces, laying out data-entry screens, reading in data from external database systems, using serial communications, and creating language parsers. Tools are also provided for manipulating Turtle Graphics, a graphics package that was included with early versions of Turbo Prolog.

In-depth descriptions of each tool and its uses are given in the *Turbo Prolog Toolbox Manual*. Because not all people who read this book will have access to the Toolbox, only a brief introduction to its use will be given here. To get you acquainted with the Toolbox—and give you some insight into its usefulness—this section introduces some of the tools and describes how they can be put to use.

Since most of the Toolbox is Turbo Prolog source code, you need only add the necessary files to your programs, and you're ready to call the predicates as you need them.

Menus

Probably, the most used part of the Toolbox is the *user interface (UI)* tool section. UI tools consist of menuing predicates, status-line predicates, input predicates, and various other tools that aid in the creation of easy-to-use programs.

UI tools are made up of different files that are added to your programs. The two Toolbox files TDOMS.PRO and TPRED.S.PRO usually get included, too, because most of the tools in Toolbox make use of the declarations made in these files. The file TDOMS.PRO contains special domain declarations and the file TPRED.S.PRO contains a general group of predicates that are referenced by many of the tools in the Toolbox.

To use a specific tool from the Toolbox, first be sure that you have included all the necessary files that are needed to let that tool

run. Since the menu tools need to use the code found in the TPRED.S.PRO and TDOMS.PRO files, these two files must be included in your programs when you use any of the menuing tools. Remember that care must be taken to add any domain or predicate declaration before it is used in the body of your code. Because of this, the file TDOMS.PRO must always be included before TPRED.S.PRO is added.

Here is a sample program showing how the predicate menu() from the Toolbox can be used:

```
* This is an example program showing the use of
* MENU.PRO from the Turbo Prolog Toolbox.
*/

include "TDOMS.PRO"    % special domains used with the Toolbox
include "TPRED.S.PRO"  % special predicates used with the Toolbox

include "MENU.PRO"      % the Menu tool from the Toolbox

Domains
    customer = c(first, last)
    first, last, id = string

Database
    client(customer, id)

Predicates
    run
    process_choice ( integer )
    delete_client ( integer )
    print_clients

Clauses
    run :-
        repeat ,      % Repeat is defined in TPRED.S.PRO
        menu(5,5,4,4,["Add Customer","Delete Customer" ,
            "Print Customers","Save Database","Quit"] ,
            " Customer Menu ",1,Choice) ,
        process_choice(Choice) ,
        Choice >= 5.    % Succeed when Quit is selected

    process_choice(1) :- ! ,
        makewindow(11,3,2," Add A Client ",3,3,10,74) ,
        write("Enter clients first name: ") ,
        readln(First) ,
        write("Enter clients last name: ") ,
        readln>Last) ,
        write("Enter clients ID number: ") ,
        readln>ID) ,
        assert( client(c(First>Last),ID) ) ,
        removewindow.

    process_choice(2) :- ! ,
        makewindow(12,4,3," Delete A Customer ",3,3,10,74) ,
        menu(7,7,2,1,["Delete by Name","Delete by ID"],
            " Delete Customer ",1,Choice) ,
        delete_client(Choice).
```

```

process_choice(3) :- ! ,
    makewindow(13,4,5," Print Customers ",3,3,10,74) ,
    write("\nPrinting Customers...") ,
    writedev(device(printer)) ,
    print_clients ,
    writedev(device(screen)) ,
    removewindow.
process_choice(4) :- ! ,
    makewindow(14,5,4," Save Customer Database ",3,3,10,74) ,
    write("\nSaving Database...") ,
    save("Clients.DBA") ,
    removewindow.
process_choice(_).      % Succeed if Quit is selected

delete_client(1) :-
    write("Enter clients first name: ") ,
    readln(First) ,
    write("Enter clients last name: ") ,
    readln>Last) ,
    retract( client(c(First>Last),_) ) , % fail if not found
    !, write("Deleted!") ,
    readchar(_), removewindow.
delete_client(1) :- ! ,
    write("That name is not in the database!") ,
    readchar(_), removewindow.
delete_client(2) :-
    write("Enter client's ID number: ") ,
    readln(ID) ,
    retract( client(_,ID) ) , % fail if ID is not found
    !, write("Deleted!") ,
    readchar(_), removewindow.
delete_client(2) :- ! ,
    write("That ID number is not in the database!") ,
    readchar(_), removewindow.

print_clients :-
    client( c(First>Last),ID) ,
    writef("%-15 %-15 %5\n",First>Last>ID) ,
    fail ; flush(printer).

% Database of Clients
client(c("Warren","Hardmen"), j388).
client(c("Johnny","Bachar"), i493).
client(c("Linn","Hiller"), i232).
client(c("Pat","Edlinger"), r137).

GOAL
makewindow(1,2,3,"",0,0,25,80) ,
run.

```

Sound

One way to add pizzazz to your programs is to add a little bit of sound to the action. Turbo Prolog supplies two built-in predicates to activate the speaker built into your computer. The first predicate

is the most basic; it simply emits a single tone from your computer system:

```
beep()
```

The second predicate allows a little more musicianship to be incorporated into your programs, because the built-in predicate

```
sound()
```

allows you to control both the frequency of the pitch produced and the duration of the note. The predicate `sound()` uses two integer values to produce a tone. The first argument represents the duration of the note (in hundredths of a second) and the second argument specifies the frequency of the pitch produced. The short program listed next demonstrates how these two sound predicates can be put to work:

```
Predicates
    run1(integer)
    run2(integer)

Clauses
    run1(400) :- sound(5,800),!.
    run1(X) :-
        sound(4.5,X) ,
        X1 = X + 20 ,
        run1(X1) .

    run2(500) :- !.
    run2(X) :-
        sound(4.5,X) ,
        X1 = X - 50 ,
        run2(X1) .

GOAL
    run1(200) ,
    beep ,
    run2(1000) .
```

Readkey () Predicate

Another useful tool that comes with Turbo Prolog is the Prolog-defined predicate

```
readkey()
```

Defined in Turbo Prolog version 2.0 as file CH18EX08.PRO, `readkey()` is useful for reading any key from the keyboard, including function keys, arrow keys, and all the other special keys that exist (such as the escape key or the backspace key). If you haven't already done so, take a look at the file CH18EX08.PRO. You will notice that this program is built up of three different predicates; `readkey()`, `key_code()`, and `key_code2()`.

The predicate `readkey()` provides the main entrance into this tool. Called with a free variable, `readkey()` returns an object that represents a key that has been read in. A look at the domain "key" will give you an indication of the different terms that can be returned. In some cases, `readkey()` will return a compound object, whereas at other times a single symbol will be returned to describe the key that has been read.

If you call `readkey()`, then press a normal alphanumeric character (such as the letter c), `readkey()` will return a compound object such as "char('c')." Function keys are also returned as compound objects from a call to `readkey()`, using the domain "fkey" to represent that a function key has been pressed.

The usefulness of this predicate will not be apparent until you come across a need to read the keyboard in this fashion. While most input can be easily handled by the built-in input predicates, `readkey()` comes in handy when you want to customize the interface of a program. As an example, suppose you want to provide support for arrow keys or function keys from your program. You'll use `readkey()` to read characters from the keyboard, then process the characters returned accordingly. Here's an example of using `readkey()` to bring up a help file from a data-entry screen:

```
/* Using the READKEY Predicate */

include "READKEY.PRO"      % Turbo Prolog v2.0 program CH18EX08.PRO

Domains
    row, col = integer
    name = c(first, last)
    first, last, id = string

Database
    client(name, id)

Predicates
    check_col ( col )
    get_text ( string )
    make_screen
    new_row ( row, row )
    process_key ( key )
```



```

repeat
run

Clauses
run :-
    makewindow(2,4,5,"",4,4,17,72) ,
    make_screen ,
    repeat ,
    readkey(Key) ,
    process_key(Key) ,
    Key = esc.

make_screen :-
    shiftwindow(2) ,
    field_str(2,2,12,"First Name: ") ,
    field_str(4,2,12,"Last Name : ") ,
    field_str(6,2,12,"ID Code   : ") ,
    field_attr(12,2,45,2) ,
    field_str(12,2,45,
        "Press ESC to end data entry      F1 for help") ,
    cursor(2,14).

process_key(fkey(1)) :- ! ,
    makewindow(10,3,2," Help ",5,30,15,40) ,
    get_text(Help_text) ,
    display(Help_text) ,
    removewindow.

process_key(char(C)) :- ! ,
    write(C).

process_key(cr) :- ! ,
    cursor(Row,_) ,
    new_row(Row,New_row) ,
    cursor(New_row, 14).

process_key(bdel) :-
    cursor(Row,Col) ,
    Col1 = Col - 1 ,
    check_col(Col1), ! ,
    field_str(Row,Col1,1," ") ,
    cursor(Row,Col1).
process_key(bdel) :-
    beep, fail.

process_key(esc) :-
    removewindow ,
    write("\n Saving Database...") ,
    save("CUSTOMER.DBA"), nl.

new_row(2,4) :- !.
new_row(4,6) :- !.
new_row(6,2) :-
    field_str(2,14,15,First) ,
    field_str(4,14,15,Last) ,
    field_str(6,14,8,ID) ,
    fronttoken(First, Fname,_) ,
    fronttoken(Last, Lname,_) ,
    fronttoken(ID, ID_1,_) ,
    assert( client(c(Fname,Lname),ID_1)) ,
    clearwindow,
    make_screen.

```

```

check_col(C) :- C >= 14.

get_text(Text) :-
    T1 = "This is the help screen!\n\n\n" ,
    T2 = "Customer names will be asserted into\nthe database " ,
    T3 = "after the customer's ID\nis entered.\n\nThe " ,
    T4 = "database will be saved\nafter you press ESC to " ,
    T5 = "exit\n\n\n(Press ESC to exit this screen)" ,
    concat(T1,T2,I1) ,
    concat(I1,T3,I2) ,
    concat(I2,T4,I3) ,
    concat(I3,T5,Text).

repeat. repeat :- repeat.

GOAL
makewindow(1,2,3," Readkey ",0,0,25,80) ,
run.

```

If the coding of the `readkey()` predicate is confusing, it may help to know that some of the keys on your keyboard actually produce two different characters when you depress the key. The keys with this property are taken care of with the definition of `readkey2()`. First, one character is read to get into this predicate, then another is read to determine the actual key typed.

14 *An Advanced Application*

When learning a new computer language, the first thing to tackle is the new language's syntax. Once the "grammar" is understood, you can put the language to work by building your own ideas into a workable program. However, simply learning the syntax is not enough to start creating programs on your own. You must also understand how the syntax can be put together to create a program that makes logical sense. In this chapter, an "advanced" program is presented and its workings are explained in detail. With a little luck (and some hard study), the techniques presented here can be adopted and used in your own Prolog programs.

The program presented is called `LOTTERY.PRO`. Used to produce "statistical" information pertaining to state lottery games, the program is a good tool for showing how the techniques introduced throughout this book can be put to use. Many different areas of Turbo Prolog are covered in this program—from list handling to writing and reading files—and that is its purpose. It is in no way intended as a tool for actually calculating the winning numbers of lottery games. Rather, the program should be looked upon as a learning aid that can be used to attack the problems you encounter when designing Prolog programs of your own.

The lottery program is divided into three separate sections. The first section gets the lottery data ready for use, the second calculates statistics based on this data, and the third creates the report from the calculations made. A quick glance at the `LOTTERY.PRO`, shown in its entirety at the end of the chapter, may prove to be overwhelming. Don't let the size throw you off because, taken in sections, the program is not difficult to follow.

Getting Started

If you wish, begin by typing in the program shown at the end of the chapter, being careful not to make mistakes (remember, comments are optional!). After you have done so, compile and run it a few times to get a feel for how it works. Since a database is not supplied with the program, you'll have to start by entering in some imaginary data.

If you're not familiar with lottery drawings, the idea is fairly simple. A range of numbers is given for you to pick from (for example, in California the range is 1 through 49). When you play the lottery, you get to make a certain number of "picks" from within the lottery range. If you're lucky, the lottery will pick the same numbers you've selected, and you'll end up being a grand prize winner!

In creating a new lottery database, you'll be prompted for the range of numbers in the lottery and the number of picks that each drawing will select (this information comprises what is known as the lottery's *dimensions*). After this, you'll be asked to enter the data for the lottery drawings. This is done by entering the numbers picked for each of the drawings that have taken place.

The statistics calculated are based on the idea of *interval performances*, or how often a number has appeared in past drawings. A lottery number "performs" with an interval of 1 if the number has been picked in two consecutive drawings—an interval of 1 means that it took one drawing for the number to be repeated. To perform with an interval of 2, a number will have been selected in two drawings, separated by one drawing in between. For example, a number appearing in drawings 3 and 5 will have a performance at

interval level 2 because after drawing 3, it took two more drawings for the number to be picked again, thus creating a performance.

The LOTTERY.PRO program generates a report based on the number of performances each lottery number has made. After the statistics are calculated, the report is first written to disk and then to the printer if a hardcopy is desired.

Glancing at the Code

A quick look at the code will give you an idea of how the program is organized. First, you'll notice a small Constants section, followed by the program's Domains section. After the Domains section, you'll find two Database sections. The first Database section is named "drawings", while the second one defaults to the name of "dbasedom", as generated by the system.

After these initial program sections, you will find several groups of Predicates and Clauses sections. Each set relates to a different task that the program performs. The first Clauses and Predicates group is the section that holds the program's *supporting predicates*. Here, you'll find general procedures that are used throughout the other sections of the program. Grouping these predicates into a single Clauses section makes it easy to add and delete general-use predicates, and allows you to see which predicates are available for general use.

Notice that the domains for the predicates in this section consist mostly of standard domain types. This allows these predicates to be available for general use without worrying about type clashes. Remember, a value representing a user-defined domain can be passed into any argument that is declared as a standard domain of the same type. Type clashes occur when a value from one user-defined domain is passed into an argument that is declared to be from a different user-defined domain.

When breaking a program up into different Predicates and Clauses sections (as this program does), it is important for the declaration of a predicate to come before the predicate is actually called in the body of the code. Because of this, the supporting predicates must be declared at the top of the code, so that they can be called

from any of the other clauses in the program.

Looking closely at the code, you'll probably notice that there's only one Domains section in the program. In larger programs, the local user-defined domains can generally be grouped into one section, since most of the predicates throughout the program make use of the same domain declarations.

Also, notice that each Predicates section of the program lists only the predicates that are defined in the following Clauses section. The ability of Turbo Prolog to include and compile several groups of predicates makes it possible for several people to work on a single project at one time. If more than one person is working on a single program, just be sure that each person involved is using the same Domains and Database sections. Doing so will ensure that separate sections of code will fit in well with the other sections written.

The last section in the program is the Goal section. A quick review reveals the order in which the program performs the task it was designed to do. Starting from the top, the program first creates a window, then consults a database into RAM. From here, information is gathered concerning the data and the report is calculated. Once calculated, all that is left to do is to output the findings.

Supporting Predicates

The first predicates listed in the program are ones that are used throughout the program. Supporting predicates range from list-handling routines to procedures that check for valid data entry. The first predicate is a member of the latter group:

```
get_answer :-  
    readchar(Ans) ,  
    write(Ans), nl ,  
    upper_lower(Ans, 'y').
```

This predicate simply checks to see if the character you have typed is equal to the letter y. Either an upper- or lowercase y is accepted, and the predicate succeeds if either is entered. If any other character is entered, the predicate fails.

Another predicate found in this section is the `rename_old__`

database() predicate. As shown in the following program code, the predicate takes a DOS file name as a string argument. The predicate checks to see if the the DOS file exists. If it does, the file is renamed to a backup file. If the file cannot be found, the predicate simply does nothing and succeeds.

```
rename_old_database(DbName) :-
    existfile(DbName), !,
    fronttoken(DbName, FileName, _) ,
    concat(FileName, ".BAK", OldDatabase) ,
    renamefile(DbName, OldDatabase) ; true.
```

This predicate is very useful when you are creating a new data file but want to save the old one as an archive copy. A call to `rename_old_database()` ensures that your old database file is not written over before a new file is created. Since this predicate is called from several different sections of the program, it has been added to the general-purpose predicates, making it available to be called from anywhere within the program.

Bringing in the Data

The second group of predicates and clauses is devoted to getting the data ready. Located under the title "Initialization Predicates," the first set of clauses in this section, `consult_db()`, allows you to select a database; it then consults the chosen database into RAM.

```
consult_db(DbName) :-
    makewindow(5,3,2,"",16,5,7,50) ,
    write("Press ESC to create a new file.") ,
    makewindow(2,4,2,"",5,5,10,50) ,
    dir("", "*.DBA", DbName) ,
    !, removewindow ,
    clearwindow ,
    write("\n\nConsulting Database") ,
    consult(DbName, drawings) ,
    removewindow.

consult_db(DbName) :-
    removewindow ,
    framewindow(2," Lottery Statistics ",-1,single_frame) ,
    get_database_name(DbName) ,
    get_lotto_max(Max) ,
    get_lotto_picks(Picks) ,
    assert( lotto_stats(Max, Picks) ) ,
    removewindow.
```


The program allows you to select a set of lottery drawings by choosing a specific database file. The built-in predicate `dir()` is ideal for displaying the appropriate database files. Taking a DOS directory and file mask as arguments, the files that fit the description are displayed in the current window. From here, the desired file can be easily chosen, then consulted into RAM.

You are also given the ability to create a new database file. By pressing `ESC` while viewing the files listed by `dir()`, the predicate `dir()` will fail. From here, you are prompted to enter a file name for the new database. To complete the file name, the file extension of `.DBA` is concatenated onto the end of the file name entered.

After a new database name is created, the vital dimensions of the database must be recorded. You are prompted for the range of numbers in the lottery and the number of picks that each drawing will select. These lottery dimensions are then saved asserted into database fact titled `lotto_stats()`. Therefore, since there is only one `lotto_stats()` per lottery database, the database fact is declared as being deterministic.

Once a database of drawings is selected, the program continues by calling the predicate `add_new_drawing()`. Here, you are able to add new lottery drawings into the database until there are no more to add, or until you get tired of making up lottery picks. After this, the lottery database is complete and the program can prepare for the final calculations.

The first thing that needs to be done is to find out the last drawing number in the database. Obtaining a correct value for the number of drawings is especially important because, as you will see, this number is used as a counter value throughout the rest of the program. The predicate `find_last_drawing_number()` does a good job of finding the last drawing number, using backtracking to step through the entire database of drawings. The predicate returns with the value corresponding to the last drawing number in the database:

```
find_last_drawing_number(_) :-
    asserta( last_num(0) ), fail.

find_last_drawing_number(_) :-
    drawing(X,_),
    asserta( last_num(X) ),
    fail.
```

```
find_last_drawing_number(X) :-  
    last_num(X) ,  
    retractall( last_num(_) ), !.
```

Because the drawings are arranged sequentially in the database, the only way to get to the last drawing number is by stepping through each database fact in turn. Saving the current high drawing number is accomplished by using a database fact titled `last_num()`.

While stepping through the `drawing()` database, each drawing number encountered is saved on top of a database stack. Using `asserta()` to enter the numbers into the database, the numbers read will always be placed on top of the previous one encountered. When at last no more `drawing()` facts are in the database, the highest number will be on the top of the `last_num()` database stack. From here, it is easy to come along and pluck this number off the top of the `last_num()` database.

Once the highest number is retrieved, the database describing `last_num()` is completely removed with a call to `retractall()`. This creates a clean operation, because no trace of the `find_last_drawing_number()` task is left behind.

After the highest drawing number is computed, all the data needed to calculate the interval statistics is completely in place and steps can now be taken to calculate the final results. From the Goals section, the next predicate in line is now called, which brings the program into the next group of predicates and clauses.

Calculating the Intervals

Now that the data is ready, the fun can begin. The first order of business is to obtain the settings of the report to be generated. First, you are free to choose the number of drawings from the database you want to use. Also, the number of intervals to be used in the calculations must be specified, as it is an important factor in the outcome of the report generated.

When you are prompted for the number of drawings to use, the

number you input is checked against the total number of drawings in the database. Obviously, you can't make use of more drawings than are available in the database. Also, notice that the maximum interval number is one less than the total number of drawings in the calculations. This stems from the fact that a database containing only two drawings can have a maximum interval level of 1. Extending this limitation shows that any interval level used must be less than the total number of drawings used in the calculations.

Once the number of drawings to use has been confirmed, the database of drawings is adjusted to reflect this setting. The predicate `retract_drawings()` does just this.

```
retract_drawings(0) :- !.

retract_drawings(X) :-
    retractall( drawing(X,_), drawings ) ,
    X1 = X - 1 ,
    retract_drawings(X1).
```

Normally, you will want to calculate the interval statistics based on a consistent number of drawings, say 100. However, until you can get a full set of drawings in your database, you'll have to use everything you've got! The predicate `retract_drawings()` provides for this. Starting off, an integer value equal to the number of drawings to retract is passed to `retract_drawings()`. This value is calculated by subtracting the number of drawings to use from the total number of drawings in the database. If all the drawings are to be used, then this value will equal zero and no drawings will be removed from the database. This relationship can be seen in the first clause defining `retract_drawings()`. When a zero is passed, `retract_drawings()` does nothing.

If, on the other hand, you do not want to use all of the available drawings, the oldest drawings in the database will be retracted until the correct number of drawings remain. Notice that retracting the old drawings at this point in no way affects the actual database of drawings stored on disk. Since the facts retracted are only removed from RAM, the database stored on disk is kept intact, and a full set of lottery drawings is still available.

A close inspection of `retract_drawings()` reveals that `retractall()` is used to retract each unwanted drawing. Although `retractall()` abolishes all occurrences of any matching database facts, this fea-

ture is not important because you know that there will only be one occurrence of each matching `drawing()` fact. An important characteristic of `retractall()`—and the reason that it is used at this point—is that it is deterministic. If `retract()` were to be used in its place, unnecessary backtracking points would be left behind, leaving room for the unexpected to take place.

Tallying the Lottery Numbers

The lottery database is now completely set up and the manipulation of the numbers can now take place. The first statistics to be computed will determine the set of drawings that each lottery number was selected in. This step creates one database fact for each number in the lottery range. The facts created will consist of a lottery number, followed by a list of the drawings in which the number was chosen. The predicate `tally_numbers()` performs this task by starting at zero and working up until all the lottery numbers have been tested against each of the drawings in the database.

```
tally_numbers(Max_number) :-
    lotto_stats(Max_number,_), !.

tally_numbers(X) :-
    Y = X + 1,
    tally_y(Y),
    tally_numbers(Y).
```

When `tally_numbers()` returns, a set of `drawing_stats()` facts will exist in the database. Tallying the lottery numbers is done by implementing a nested loop. The first loop in the sequence steps through each possible lottery number. With each lottery number, a second loop is called to calculate the drawings in which that number was selected.

The predicate `tally_y()` takes care of the second loop. Using backtracking, each fact in the `drawing()` database is looked up and tested to see whether the lottery number being processed was picked in that drawing. A call to `member()` performs this test. If the call to `member()` succeeds, then the lottery number being processed was

chosen in the drawing just looked up, and the drawing is then added to the list associated with the current lottery number.

While the description of this process is a bit wordy and hard to understand, the Prolog definition of `tally_y()` describes this processing in a more concise fashion:

```
tally_y(Y) :-
    drawing(Num, List) ,           % Look up a drawing number and
                                   % its list of lottery picks.

    member(Y,List) ,              % Is the current lottery number
                                   % being processed in this lottery
                                   % drawing.

    add_to_list(Y,Num) ,           % If so, add the drawing number to
                                   % the drawing list associated with
                                   % the current lottery number.

    fail ; true.                  % Repeat until done, finding all
                                   % the lottery drawings where the
                                   % current lottery number was
                                   % picked.
```

Next in line is the call to `add_to_list()`. This predicate takes a drawing number and adds it to the list of drawings in which a particular lottery number was picked. Compared with `tally_y()`, the Prolog definition for `add_to_list()` may look complicated. However, it is actually easy to understand:

```
add_to_list(Y,Num) :-
    drawing_stats(Y,List,Repeats) , ! ,
    make_list(Num, List, New_list) ,
    retractall( drawing_stats(Y,List,Repeats) ) ,
    Repeats_1 = Repeats + 1 ,
    assertz( drawing_stats(Y,New_list,Repeats_1) ) .

add_to_list(Y,Num) :- assertz( drawing_stats(Y,[Num],1) ) .
```

This procedure starts off by checking to see whether the lottery number being processed has a database fact associated with it. This is done by calling `drawing_stats()` with the respective lottery number. If the lottery number being processed is already associated with a `drawing_stats()` fact in the database, then that fact is updated by adding the new drawing number just found.

If no match is found in the database for the current lottery number, then the call to `drawing_stats()` will fail and a new database fact will have to be created for that lottery number. When the

new fact is created, the drawing number is attached to the lottery number by making a list consisting of the one drawing number. In addition to the list, the fact contains a counter to keep track of the number of drawings in the list. This counter is set to 1 in this first fact, which correctly reflects the number of drawings in the list. The call that asserts all of this information is represented with the line

```
assertz( drawing_stats(Y,[Num],1) )
```

Once this fact is in the database, `add_to_list()` can check for its existence when a new drawing number is found, and any new drawing numbers can then be added to the list.

Tallying the drawing numbers may seem complex at first glance. However, the procedure is created by putting two types of looping structures together. Note that `tally_numbers()` is made up from a simple recursive loop. The terminating condition is dynamic, meaning that it can change from run to run of the program. Since the range of lottery numbers fluctuates from one state lottery to another, this condition can't be hard coded into the program, so a dynamic condition must be set up.

The dynamic test can be seen in the first clause defining `tally_numbers()`. Here the lottery number being processed is compared with the greatest amount of numbers allowed by the range of the lottery. If the two numbers match, the processing is finished and a return from `tally_numbers()` can be made.

A second loop is encountered in the predicate `tally_y()`. This loop is created by forcing backtracking to process all of the `drawing()` facts in the database. When there are no more `drawing()` facts in the database to process, `tally_y()` is finished and returns after a call is made to the predicate `true()`.

Calculating the Intervals

The next part of the program is the piece that actually produces the interval statistics. From the Goal section, the predicate `lotto_stats()` is called to obtain the value for the maximum number in the

lottery range. This value is then passed to the predicate `calculate_intervals()`, along with the number of intervals that are to be calculated. In the predicate `calculate_intervals()`, a loop is set up using the number of intervals to calculate, to initialize (seed) the counter.

From inside `calculate_intervals()`, the predicate `calc_number_intervals()` is called to process the interval calculations. After the return is made from `calc_number_interval()`, the interval number is decremented by one and the `calculate_intervals()` loop is started over again. When the interval number finally hits zero, all the intervals have been calculated and the looping can stop.

```
calculate_intervals(0,_) :- !.

calculate_intervals(Interval, Max) :-
    calc_number_interval(Interval, Max) ,
    Int_1 = Interval - 1 ,
    calculate_intervals(Int_1, Max).
```

With each interval number obtained in `calculate_intervals()`, a call is made to the predicate `calc_interval_number()`. Here, another loop is set up, this time using the maximum lottery number as the value that sets the counter. In this loop, each lottery number is tested for the current interval value being processed. A lottery number is looked up (with a call to `drawing_stats()`), and the list of drawings in which the lottery number was picked is retrieved. Using this information, a call to `count_performance()` is made to do the actual counting of the intervals.

In `calc_number_interval()`, the case must be supported for when no `drawing_stats()` fact is associated with a particular lottery number. This must be accounted for because when there are only a few drawings in the database it is likely that not every lottery number in the range will have been picked. For this reason, `calc_number_interval()` has a third clause, which fires when a lottery number is not represented by a `drawing_stats()` database fact. In this case, the lottery number to process is decremented by 1, and the loop starts over again from the top.

```
calc_number_interval(_, 0) :- !.

calc_number_interval(Int, Number) :-
    drawing_stats(Number, List, _), ! ,
    count_performance(Number, List, Int, 0) ,
    Number_1 = Number - 1 ,
    calc_number_interval(Int, Number_1).
```



```

calc_number_interval(Int, Number) :-
    Number_1 = Number - 1 ,
    calc_number_interval(Int, Number_1).

```

The last procedure called in this sequence is the predicate that does the actual calculating of the performance for each lottery number. Up to now, two loops have been set up. The first is a loop that steps through each interval level. Inside this loop, another loop is called to go through each lottery number, calculating its performance for each interval level in the report. Besides these two loops, all that is needed to finish the calculations is a procedure that calculates the performance of each lottery number for the current interval level being processed.

```

count_performance(Num, [_], _, Repeats) :-
    retract(interval(Num, Performance)), ! ,
    New_performance = Performance + Repeats ,
    asserta(interval(Num, New_performance)).

count_performance(Num, [_], _, Repeats) :-
    asserta( interval(Num, Repeats) ), !.

count_performance(Num, [H|T], Int, Repeats) :-
    Performance = H - Int ,
    member(Performance, T), ! ,
    Repeats_1 = Repeats + 1 ,
    count_performance(Num, T, Int, Repeats_1).

count_performance(Num, [_|T], Int, Repeats) :-
    count_performance(Num, T, Int, Repeats).

```

The predicate `count_performance()` does just this. The predicate is called with a lottery number, a list of drawings in which that number was picked, the current interval number, and a performance counter (which was initialized to zero in the original call). Given this information, `count_performance()` steps through each element in the list of drawings, testing for interval performances. With each step through the loop, a test is made to see if the number at the top of the list proves to be an interval performance. If it so happens that the head of the list is a performing number, then the performance counter is incremented by 1, and the loop is called again with the tail of the drawings list.

A performance is calculated by subtracting the current interval level from the drawing value in the head of the drawings list. When this calculation is made, a target interval number is obtained. If this target number can be found to exist in the tail of the drawings

list, then an interval performance is found between the head of that list and the target value calculated. Testing for the target value is done with a call to the predicate `member()`, where the tail of the drawings list is tested to see if it contains the target number calculated.

If no performance is found, then the call to `member()` fails and the next clause of `count_performance` must be called. In either case, the loop starts over again from the top, this time with only the tail of the original drawings list.

The looping in `count_intervals()` continues until the drawings list has only a single element in it (a list with only one element cannot possibly have an interval performance associated with it). Once the last element in the list is reached, a test is made to see if the current lottery number has had any other performances associated with it. If it has, the new performance totals are added to the existing ones, and the new total is asserted into the database. If no fact is associated with the current lottery number being processed, then a new fact is asserted into the `interval()` database for the lottery number being processed.

Again, the processing of these Prolog clauses becomes wordy when trying to detail all the steps that the code represents. Take your time to study the parts that seem cloudy. After practicing, you'll find that the code written in Prolog actually creates a good description for the processing that needs to take place. It's no coincidence that Prolog is considered a descriptive computer language.

Writing Out the Results

After all the lottery statistics are tallied and computed, the only thing left to do is to write them out so that they can be inspected. During the processing, `LOTTERY.PRO` creates two sets of results, both of which are written to a disk file named `RESULTS.DBA`. Once the final report is completed and written to disk, you have the option of printing out the report. To produce a hard copy, reopen the report file and read the results to the printer.

In the program, the first set of statistics is calculated when a call to `tally_numbers()` is made. The predicate `tally_numbers()` creates a set of facts that details the drawings in which each lottery number was picked. In addition to a list of drawing numbers, the total number of times each lottery number was chosen is also stored as a part of the database facts.

The first part of the report outputs this information in a readable format. The predicate `output_tallies()` shows how a new file is opened and written to:

```
output_tallies :-
    rename_old_database("RESULTS.DAT") ,
    openwrite(resultfile, "RESULTS.DAT") ,
    writedevise(resultfile) ,
    write_tallies ,
    flush(resultfile) ,
    closefile(resultfile) ,
    writedevise(screen).
```

Once the file `RESULTS.DAT` is opened for writing, `write_tallies()` is called to do the actual dirty work. Creating a loop with the use of backtracking, each `drawing_stats()` fact is looked up one at a time. The data obtained from each fact is formatted and written to the file `RESULTS.DAT`. When this loop has finished, the result file is closed and the output stream is returned to its normal screen setting.

Closing the `RESULTS.DAT` file is not necessary at this time, since the very next predicate reopens it to write more data. However, closing and opening the file is shown to illustrate how the predicate `openappend()` can be used to tack information onto the end of an existing file. After the tallies have been written to the report file, the file is reopened (this time with `openappend()`), and the interval statistics are appended to the end of the report.

After the tallies are written out, `report_intervals()` is called from the Goal section to print out the final interval statistics. This predicate acts as a driver predicate (much like `output_tallies()` does), by simply setting things up for the actual writing out of data. The predicate `report_intervals()` is much like the predicate `output_tallies()`. In `report_intervals()`, the report file is opened for writing, the output stream is redirected, and a call is made to write the report. After the reporting is finished, the report file is closed and the output stream is set back to normal.

To make the best report, the lottery numbers are printed in descending order, starting with the number that has the highest interval performance. To do the actual printing, `print_intervals()` calls `print_report()` with an initial value of 100. The value of 100 assures that all the statistics are printed out, assuming that no lottery numbers have an interval performance higher than 100. The following code prints the report:

```
print_report(-1) :- !.

print_report(Performance) :-
    retract(interval(Num,Performance)), !,
    writef("The number %2 performed %2 times.\n" ,
          Num, Performance) ,
    print_report(Performance).

print_report(Performance) :-
    Performance_1 = Performance - 1 ,
    print_report(Performance_1).
```

First, `print_report()` checks to see if any lottery numbers have an interval performance of 100. If a number is found that matches this criterion, a report is generated that details the lottery number and its interval performance (which would be 100 at this point). The fact describing the lottery number and interval is retracted from the database, and `print_report()` is called again with the same performance number.

If no lottery numbers can match the current interval number being processed, then the call to

```
retract(interval(Num,Performance))
```

will fail, and the last clause of `print_report()` will be called. In this clause, the performance number is decremented by one, and the report process is called again with the new interval number. Notice that only when no more lottery numbers are associated with the current performance number being processed is the performance number decremented. This way, all lottery numbers with the same performance can be looked up and added to the report. When there are no more lottery numbers with the current performance number, the next lower performance number is tried and the reporting process starts over.

This loop continues until the interval performance dips below zero. The value -1 is used as a stopping condition to the loop because there is a good chance that some lottery numbers will have a zero performance value. If the loop stopped at zero (instead of -1), all of these numbers would be skipped in the final report.

Once all the statistics are written to the file RESULTS.DAT, processing comes back to the Goal section and a call is made to the predicate `report_2_printer()`. In this call, you are asked if you want to print out a copy of the report just generated. If you do, the output stream is switched over to the printer and the file RESULTS.DAT is opened for reading. The file RESULTS.DAT is then output to the printer, one line at a time. When the printed report is finally completed, the call to `report_2_printer()` can make its return. From here, the final subgoal in the Goal section can be called.

Ending with a simple write statement, the lottery program finishes its execution by writing "Done!" The entire program now terminates, and you're returned to the place where you started executing LOTTERY.PRO.

```

/*  LOTTERY.PRO  */

/*
 * This program computes lottery "statistics",
 * based on the frequency that the lottery numbers
 * have been drawn. The "performance" of a number
 * is based upon the number of times that a lottery
 * number is repeated within a given interval of drawings.
 */

Constants
    single_frame = "\218\191\192\217\196\179"

Domains
    list = integer*
    file_name = string
    save_flag = on ; off
    drawing_number, repeats ,
        number_drawn, number_of_picks, interval = integer
    file = resultfile

Database - drawings
    drawing ( drawing_number , list )
    Determ lotto_stats ( number_drawn, number_of_picks )

Database
    Determ last_num ( drawing_number )
    drawing_stats ( number_drawn, list, repeats )
    interval ( number_drawn, repeats)

```


Predicates

```

/* * * * * *
 * Supporting Predicates *
 * * * * * */

get_answer
make_list ( integer, list, list )
Determ member ( integer, list )
read_integer ( integer )
rename_old_database ( file_name )

```

Clauses

```

get_answer :-
    readchar(Ans) ,
    write(Ans), nl ,
    upper_lower(Ans, 'y').

make_list(E,L,[E|L]).

member(H,[H|_]) :- !.
member(H,[_|_]) :- member(H,T).

read_integer(I) :-
    readint(I), ! ;
    read_integer(I).

rename_old_database(DbName) :-
    existfile(DbName), ! ,
    fronttoken(DbName, FileName, _) ,
    concat(FileName, ".BAK", OldDatabase) ,
    renamefile(DbName, OldDatabase) ; true.

```

Predicates

```

/* * * * * *
 * Initialization Predicates *
 * * * * * */

add_new_drawing ( drawing_number, file_name, save_flag )
consult_db ( file_name )
find_last_drawing_number ( drawing_number )
get_database_name ( file_name )
get_lotto_max ( number_drawn )
get_lotto_picks ( number_of_picks )
get_new_drawing ( number_of_picks, list )

```

Clauses

```

consult_db(DbName) :-
    makewindow(5,3,2,"",16,5,7,50) ,
    write("Press ESC to create a new file.") ,
    makewindow(2,4,2,"",5,5,10,50) ,
    dir("", "*.DBA", DbName) ,
    !, removewindow ,
    clearwindow ,
    write("\n\nConsulting Database") ,
    consult(DbName, drawings) ,
    removewindow.

```

```

consult_db(DbName) :-
    removewindow ,
    framewindow(2," Lottery Statistics ",-1,single_frame) ,
    get_database_name(DbName) ,
    get_lotto_max(Max) ,
    get_lotto_picks(Picks) ,
    assert( lotto_stats(Max, Picks) ) ,
    removewindow.

get_database_name(DbName) :-
    clearwindow ,
    write("\nEnter the database name: ") ,
    cursor(R,C) ,
    write("\n\n (Do not enter a file extension name!)" ) ,
    cursor(R,C) ,
    readln(Name) ,
    concat(Name, ".DBA", DbName).

get_lotto_max(Lotto_max) :-
    clearwindow ,
    write("\nEnter the highest possible lottery number: ") ,
    read_integer(Lotto_max).

get_lotto_picks(Lotto_picks) :-
    clearwindow ,
    write("\nEnter the number of picks in each drawing: ") ,
    read_integer(Lotto_picks).

add_new_drawing(High_drawing, DbName, _) :-
    write("\nDo you wish to add a new drawing " ,
        "to the database? (y/n) ") ,
    get_answer, ! ,
    New_high = High_drawing + 1 ,
    get_new_drawing(0, Drawing) ,
    assertz( drawing(New_high, Drawing) ) ,
    clearwindow ,
    add_new_drawing(New_high, DbName, on).
add_new_drawing(_, DbName, on) :-
    clearwindow ,
    write("\nWould you like to save the new database? (Y/N) ") ,
    get_answer ,
    write("\n\nSaving Database..." ) ,
    rename_old_database(DbName) ,
    save(DbName,drawings) ,
    fail.
add_new_drawing(_,_,_) :-
    clearwindow.

get_new_drawing(X,[]) :- lotto_stats(_,X), !.
get_new_drawing(Num,[H|T]) :-
    Num1 = Num + 1 ,
    writef("\nEnter a number that was drawn (%): ", Num1) ,
    readint(H) ,
    get_new_drawing(Num1, T).

find_last_drawing_number(_) :-
    asserta( last_num(0) ), fail.
find_last_drawing_number(_) :-
    drawing(X,_) ,
    asserta( last_num(X) ) ,
    fail.

```

```
find_last_drawing_number(X) :-
    last_num(X) ,
    retractall( last_num(_) ), !.
```

Predicates

```
/* * * * * *
 * Calculation Predicates *
 * * * * * */

add_to_list ( number_drawn, drawing_number )
calculate_intervals ( interval, number_drawn )
calc_number_interval ( interval, number_drawn )
count_performance(number_drawn, list, interval, repeats )
get_drawings_2_use ( drawing_number, drawing_number )
get_intervals ( interval, drawing_number )
get_report_settings ( drawing_number, interval, drawing_number )
retract_drawings ( drawing_number )
tally_numbers ( number_drawn )
tally_y ( number_drawn )
```

Clauses

```
get_report_settings(High_drawing, Intervals, Drawings_2_use) :-
    writef("There are % drawings in the database.\n" ,
           High_drawing) ,
    get_drawings_2_use(Drawings_2_use, High_drawing) ,
    get_intervals(Intervals, Drawings_2_use).

get_drawings_2_use(Drawings, High_drawing) :-
    write("\nHow many drawings back would you like " ,
           "to analyze? ") ,
    read_integer(Drawings) ,
    Drawings <= High_Drawing, !.
get_drawings_2_use(Drawings, High_drawing) :-
    beep ,
    write(" There aren't that many drawings in " ,
           "the database!\n") ,
    get_drawings_2_use(Drawings, High_drawing).

get_intervals(Intervals, Drawings_2_use) :-
    write("\nHow many drawing intervals would you " ,
           "like to see? ") ,
    read_integer(Intervals) ,
    Intervals < Drawings_2_use, !.
get_intervals(Intervals, Drawings_2_use) :-
    beep ,
    write(" There aren't that many drawings to use!\n") ,
    get_intervals(Intervals, Drawings_2_use).

retract_drawings(0) :- !.
retract_drawings(X) :-
    retractall( drawing(X,_), drawings ) ,
    X1 = X - 1 ,
    retract_drawings(X1).

tally_numbers(Max_number) :-
    lotto_stats(Max_number,_), !.
tally_numbers(X) :-
    Y = X + 1 ,
    tally_y(Y) ,
    tally_numbers(Y).
```

```

tally_y(Y) :-
    drawing(Num, List) ,
    member(Y, List) ,
    add_to_list(Y, Num) ,
    fail ; true.

add_to_list(Y, Num) :-
    drawing_stats(Y, List, Repeats), ! ,
    make_list(Num, List, New_list) ,
    retractall( drawing_stats(Y, List, Repeats) ) ,
    Repeats_1 = Repeats + 1 ,
    assertz( drawing_stats(Y, New_list, Repeats_1) ).
add_to_list(Y, Num) :- assertz( drawing_stats(Y, [Num], 1) ).

calculate_intervals(0, _) :- !.
calculate_intervals(Interval, Max) :-
    calc_number_interval(Interval, Max) ,
    Int_1 = Interval - 1 ,
    calculate_intervals(Int_1, Max).

calc_number_interval(_, 0) :- !.
calc_number_interval(Int, Number) :-
    drawing_stats(Number, List, _), ! ,
    count_performance(Number, List, Int, 0) ,
    Number_1 = Number - 1 ,
    calc_number_interval(Int, Number_1).
calc_number_interval(Int, Number) :-
    Number_1 = Number - 1 ,
    calc_number_interval(Int, Number_1).

count_performance(Num, [_], _, Repeats) :-
    retract(interval(Num, Performance)), ! ,
    New_performance = Performance + Repeats ,
    asserta(interval(Num, New_performance)).
count_performance(Num, [_], _, Repeats) :-
    asserta( interval(Num, Repeats) ), !.
count_performance(Num, [H|T], Int, Repeats) :-
    Performance = H - Int ,
    member(Performance, T), ! ,
    Repeats_1 = Repeats + 1 ,
    count_performance(Num, T, Int, Repeats_1).
count_performance(Num, [_|T], Int, Repeats) :-
    count_performance(Num, T, Int, Repeats).

```

Predicates

```

/* * * * * *
 * Output Predicates *
 * * * * *
*/

```

```

output_tallies
print_report(repeats)
report_intervals
report_2_printer
write_tallies
write_2_printer

```

Clauses

```

output_tallies :-
    rename_old_database("RESULTS.DAT") ,
    openwrite(resultfile, "RESULTS.DAT") ,
    writedevise(resultfile) ,

```



```

    write_tallies ,
    flush(resultfile) ,
    closefile(resultfile) ,
    writedevic(screen).

write_tallies :-
    drawing_stats(Number_drawn, List, Repeats) ,
    writef("%2 repeated %2 times: ", Number_drawn, Repeats) ,
    write(List, "\n") ,
    fail ; write('\12').      % form feed

report_intervals :-
    openappend(resultfile, "RESULTS.DAT") ,
    writedevic(resultfile) ,
    print_report(100) ,
    flush(resultfile) ,
    closefile(resultfile) ,
    writedevic(screen).

print_report(-1) :- !.
print_report(Performance) :-
    retract(interval(Num, Performance)), ! ,
    writef("The number %2 performed %2 times.\n" ,
        Num, Performance) ,
    print_report(Performance).
print_report(Performance) :-
    Performance_1 = Performance - 1 ,
    print_report(Performance_1).

report_2_printer :-
    clearwindow ,
    write("Do you want to print the report? (y/n) ") ,
    get_answer, ! ,
    write("\nPrinting Report...") ,
    writedevic(printer) ,
    openread(resultfile, "RESULTS.DAT") ,
    readdevic(resultfile) ,
    write_2_printer ,
    closefile(resultfile) ,
    writedevic(screen) ,
    readdevic(keyboard).
report_2_printer.

write_2_printer :-
    not(eof(resultfile)), ! ,
    readln(Report) ,
    write(Report), nl ,
    write_2_printer.
write_2_printer :-
    write('\12'), flush(printer).

/* * * * */
GOAL
/* * * * */

% Initialize Database
makewindow(1,2,3," Lottery Intervals ",0,0,25,80) ,
consult_db(DbName) ,
find_last_drawing_number(Last_num) ,
add_new_drawing(Last_num, DbName, off) ,
find_last_drawing_number(High_drawing) ,

```

```
% Calculate Report
get_report_settings(High_drawing, Intervals, Drawings_2_use) ,
clearwindow ,
field_str(2,2,40,"Calculating Report...") ,
Unused_drawings = High_drawing - Drawings_2_use ,
retract_drawings(Unused_drawings) ,
field_str(4,2,40,"Tallying Numbers Drawn...") ,
tally_numbers(0) ,
field_str(6,2,40,"Calculating Intervals...") ,
lotto_stats(Max,_) ,
calculate_intervals(Intervals, Max) ,

% Create Report
field_str(8,2,40,"Writing Out Report...") ,
output_tallies ,
report_intervals ,
report_2_printer ,
write("\n\nDone!\n").

/* END */
```


A ***Built-in Predicates***

This appendix consists of listings of all the built-in predicates in Turbo Prolog version 2.0. The predicates are grouped into these major categories depicting their use:

BGI (Borland Graphics Interface)

Conversion

Editor

Error Control

External Database: Chains

External Database: B+ Trees

File Handling

Flow Control

Input

Internal Database

Machine Level

Output

Screen Handling

String Handling

System (DOS)

Windows

Miscellaneous

Abbreviations and Notation

Built-in predicates are represented in three ways. In the first column, predicates are shown by how they are declared internally in Turbo Prolog. The second column gives a general impression of the arguments that the predicates deal with. The third column lists the flow pattern(s) that each predicate can be used with.

Any predicate that has three arguments or more uses the abbreviations shown in the following table for the domain declarations of the predicate:

Abbreviation	Standard Domain Type
int	integer
str	string
sym	symbol
db_sel	db_selector

NOTE Turbo Prolog automatically generates the following domain declaration for use with certain BGI predicates:

```
bgi_iList = integer*
```

Notation

The following special notations are used in the listings:

- [] Anything enclosed in square brackets is optional.
- ... An ellipsis indicates that an item can be repeated.
- | A vertical bar symbolizes a "this or that" situation. For example, on|off symbolizes that on or off can be used.
- i|o This notation is used in certain flow patterns to indicate that an argument can take a partially bound value. An example is using a partially bound term in a call to retract().

BGI (Borland Graphics Interface)

Built-in Predicate Declarations	Arguments	Flow Patterns
arc(int,int,int,int,int)	(X,Y,StAngle,EndAngle, Radius)	(i,i,i,i,i)
bar(int,int,int,int)	(Left,Top,Right,Bottom)	(i,i,i,i)
bar3d(int,int,int,int,int,int)	(Left,Top,Right,Bottom, Depth,Mode)	(i,i,i,i,i,i)
Mode: 0=flag 0<>no flag		
circle(int,int,int)	(X,Y,Radius)	(i,i,i)
cleardevice		()
clearviewport		()
closegraph		()
detectgraph(integer,integer)	(BgiDriver,GraphMode)	(o,o)
drawpoly(bgi__iList)	(PointList)	(i)
ellipse(int,int,int,int,int,int)	(X,Y,StAngle,EndAngle, Xradius,Yradius)	(i,i,i,i,i,i)
fillellipse(int,int,int,int)	(X,Y,Xradius,Yradius)	(i,i,i,i)
fillpoly(bgi__iList)	(PointList)	(i)
floodfill(int,int,int)	(X,Y,BoundaryColor)	(i,i,i)
getarccoords(int,int,int, int,int,int)	(X,Y,Xstart,Ystart, Xend,Yend)	(o,o,o,o,o,o,o)
getaspectratio(integer,integer)	(Xasp,Yasp)	
getbkcolor(integer)	(BkgndColor)	(o)
getcolor(integer)	(ForgndColor)	(o)
getdrivername(string)	(BgiDriver)	(o)
getdefaultpalette(bgi__iList)	(DefaultPalette)	(o)
getfillpattern(bgi__iList)	(UserPattern)	(o)
getfillsettings(integer,integer)	(Pattern,Color)	(o,o)
getgraphmode(integer)	(GraphMode)	(o)
getimage(int,int,int,int,str)	(Left,Top,Right,Bottom, BitMap)	(i,i,i,i,o)
getlinesettings(int,int,int)	(Style,Pattern,Thickness)	(o,o,o)
getmaxcolor(integer)	(MaxColor)	(o)
getmaxx(integer)	(MaxX)	(o)
getmaxy(integer)	(MaxY)	(o)
getmaxmode(integer)	(MaxMode)	(o)
getmodename(integer,string)	(ModeNumber,ModeName)	
		(i,o)

Built-in Predicate Declarations	Arguments	Flow Patterns
getmoderange(int,int,int)	(BgiDriver,LoMode,HiMode)	(i,o,o)
getpalette(bgi__iList)	(Palette)	(o)
getpalettesize(integer)	(Size)	(o)
getpixel(int,int,int)	(X,Y,Color)	(i,i,o)
gettextsettings(int,int,int, int,int)	(Font,Direction,CharSize, Horiz,Vert)	(o,o,o,o,o)
getviewsettings(int,int,int, int,int)	(Left,Top,Right,Bottom, ClipFlag)	(o,o,o,o,o)
getx(integer)	(X)	(o)
gety(integer)	(Y)	(o)
graphdefaults		()
imagesize(int,int,int,int,int)	(Left,Top,Right,Bottom,Size)	(i,i,i,i,o)
initgraph(int,int,int,int,str)	(BgiDriver,GraphMode, NewDriver, NewMode, DriverPath)	(i,i,o,o,i)
line(int,int,int,int)	(X0,Y0,X1,Y1)	(i,i,i,i)
linerel(integer,integer)	(RelX,RelY)	(i,i)
lineto(integer,integer)	(X,Y)	(i,i)
moverel(integer,integer)	(RelX,RelY)	(i,i)
moveto(integer,integer)	(X,Y)	(i,i)
outtext(string)	(Var)	(i)
outtextxy(int,int,str)	(X,Y,Var)	(i,i,i)
pieslice(int,int,int,int,int)	(X,Y,StAngle,EndAngle, Radius)	(i,i,i,i,i)
putimage(int,int,str,int)	(X,Y,BitMap,PutOp)	(i,i,i,i)
putpixel(int,int,int)	(X,Y,Color)	(i,i,i)
rectangle(int,int,int,int)	(Left,Top,Right,Bottom)	(i,i,i,i)
restorecrtmode		()
setactivepage(integer)	(Page)	(i)
setallpalette(bgi__iList)	(Palette)	(i)
setaspectratio(integer,integer)	(Xasp,Yasp)	(i,i)
setbkcolor(integer)	(BkgndColor)	(i)
setcolor(integer)	(ForgndColor)	(i)
setfillpattern(bgi__iList, integer)	(UserPattern,Color)	(i,i)
setfillstyle(integer,integer)	(UserPattern,Color)	(i,i)
setgraphbufsize(integer)	(BufferSize)	(i)
setgraphmode(integer)	(GraphMode)	(i)
setlinestyle(int,int,int)	(Style,Pattern,Thickness)	(i,i,i)
setpalette(integer,integer)	(Index,Color)	(i,i)

Built-in Predicate Declarations	Arguments	Flow Patterns
settextjustify(integer, integer)	(Horiz, Vert)	(i, i)
settextstyle(int, int, int)	(Font, Direction, CharSize)	(i, i, i)
setusercharsize(int, int, int, int)	(MultX, DivX, MultY, DivY)	(i, i, i, i)
setviewport(int, int, int, int, int)	(Left, Top, Right, Bottom, ClipFlag)	(i, i, i, i, i)
ClipFlag : 0=no clip 0<>clip		
setvisualpage(integer)	(No)	(i)
setwritemode(integer)	(Mode)	(i)
Mode : 0=overwrite color 1=combine color		
textheight(string, integer)	(Text, Height)	(i, o)
textwidth(string, integer)	(Text, Width)	(i, o)

Conversion

Built-in Predicate Declarations	Arguments	Flow Patterns
char__int(char, integer)	(Var, Var)	(i, i) (i, o) (o, i)
str__char(string, char)	(Var, Var)	(i, i) (i, o) (o, i)
str__int(string, integer)	(Var, Var)	(i, i) (i, o) (o, i)
str__real(string, real)	(Var, Var)	(i, i) (i, o) (o, i)
upper__lower(char, char)	(Var, Var)	(i, i) (i, o) (o, i)
upper__lower(string, string)	(Var, Var)	(i, i) (i, o) (o, i)

Editor

Built-in Predicate Declarations	Arguments	Flow Patterns
display(string)	(Var)	(i)
edit(string,string)	(Input,Output)	(i,o)
edit(str,str,str,str,str,int, str,int,int,int,int, int,int)	(In,Out,Header, FileName, Msg, StPos,HelpFile, Mode,Indent,Insert, Wrap,EndPos,Exit)	(i,o,i,i,i,i,i,i,i,o,o)
Mode: 0=display 1=edit Indent: 0=auto off 1=auto on Insert: 0=overwrite 1=insert Wrap: 0=off 1=on Exit: 0=F10 1=Esc		
editmsg(str,str,str, str,str,int,str,int)	(In,Out,Header1, Header2,Msg, Pos,HelpFile,Exit)	(i,o,i,i,i,i,o)

Error Control

Built-in Predicate Declarations	Arguments	Flow Patterns
break(symbol)	(Mode)	(i) (o)
Mode = on off		
breakpressed		
consulterror(str,int,real)	(Line,PosInLine,FilePos)	(o,o,o)
criticalError(int,int,int,int) - language C	(Error,Type,Disk>Action)	(i,i,i,o)
errormsg(str,int,str,str)	(DosFile,Error,Msg, ExtraHelp)	(i,i,o,o)

Built-in Predicate Declarations	Arguments	Flow Patterns
exit		
exit(integer)	(ExitCode)	(i)
fileError(integer,string) - language C	(Error,DosFile)	(i,i)
readtermerror(string,integer)	(Line,PosInLine)	(o,o)
trap(trap__predicate,error__ code,error__predicate)		(i o,o,i o)

External Database: Chains

Built-in Predicate Declarations	Arguments	Flow Patterns
chain__delete(db__ selector,string)	(DbName,Chain)	(i,i)
chain__first(db__sel,str,ref)	(DbName,Chain,First)	(i,i,o)
chain__inserta(db__sel, str,domain,term,ref)	(DbName,Chain,Domain, Term,Ref)	(i,i,i,i,o)
chain__insertafter(db__sel, domain,ref,term,ref)	(DbName,Domain,Ref,Term, NewRef)	(i,i,i,i,o)
chain__insertz(db__sel, str,domain,term,ref)	(DbName,Chain,Domain, Term,Ref)	(i,i,i,i,o)
chain__last(db__sel,str,ref)	(DbName,Chain,Last)	(i,i,o)
chain__next(db__sel,ref,ref)	(DbName,Ref,Next)	(i,i,o)
chain__prev(db__sel,ref,ref)	(DbName,Ref,Prev)	(i,i,o)
chain__terms(db__sel, str,domain,term,ref)	(DbName,Chain,Domain, Term,Ref)	(i,i,i,i o,o)
db__chains(db__selector, string)	(DbName,Chain)	(i,o)
db__close(db__sel)	(DbName)	(i)
db__copy(db__sel,str,place)	(DbName,Name,Place)	(i,i,i)
db__create(db__sel,str,place)	(DbName,Name,Place)	(i,i,i)
db__delete(string,place)	(Name,Place)	(i,i)
db__flush(db__selector)	(DbName)	(i)

Built-in Predicate Declarations	Arguments	Flow Patterns
db__garbagecollect(db__selector)	(DbName)	(i)
db__open(db__sel,str,place)	(DbName,Name,Place)	(i,i,i)
db__openinvalid(db__sel, str,place)	(DbName,Name,Place)	(i,i,i)
db__statistics(db__sel, real,real,real,real)	(DbName,Len,RamSize, FileSize,AmtFree)	(i,o,o,o,o)
ref__term(db__sel, domain,ref,term)	(DbName,Domain,Ref,Term)	(i,i,i,i o)
term__delete(db__sel,str,ref)	(DbName,Chain,Ref)	(i,i,i)
term__replace(db__sel, domain,ref,term)	(DbName,Domain,Ref, NewTerm)	(i,i,i,i)

External Database: B+ Trees

Built-in Predicate Declarations	Arguments	Flow Patterns
bt__close(db__selector, bt__selector)	(DbName,BtSel)	(i,i)
bt__create(db__sel,str, bt__sel,int,int)	(DbName,Btree,BtSel, KeyLen,Order)	(i,i,o,i,i)
bt__delete(db__selector, string)	(DbName,Btree)	(i,i)
bt__open(db__sel,str,bt__sel)	(DbName,Btree,BtSel)	(i,i,o)
bt__statistics(db__sel,bt__ sel,real,real,int,int,int,int)	(DbName,BtSel,Len,Pages Depth,KeyLen,Order, PageSize)	(i,i,o,o,o,o,o,o)
db__btrees(db__selector, string)	(DbName,Btree)	(i,o)
key__delete(db__sel, bt__sel,str,ref)	(DbName,BtSel,Key,Ref)	(i,i,i,i)
key__first(db__sel, bt__sel,ref)	(DbName,BtSel,First)	(i,i,o)
key__insert(db__sel, bt__sel,str,ref)	(DbName,BtSel,Key,Ref)	(i,i,i,i)

Built-in Predicate Declarations	Arguments	Flow Patterns
key__current(db__sel, bt__sel,str,ref)	(DbName,BtSel,Key,Ref)	(i,i,o,o)
key__last(db__sel,bt__sel,ref)	(DbName,BtSel,Last)	(i,i,o)
key__next(db__sel, bt__sel,ref)	(DbName,BtSel,Next)	(i,i,o)
key__prev(db__sel, bt__sel,ref)	(DbName,BtSel,Prev)	(i,i,o)
key__search(db__sel, bt__sel,str,ref)	(DbName,BtSel,Key,Ref)	(i,i,i,o)

File Handling

closefile(file)	(SymName)	(i)
deletefile(string)	(DosFile)	(i)
disk(string)	(DosPath)	(i)
		(o)
eof(file)	(SymName)	(i)
existfile(string)	(DosFile)	(i)
filemode(file,integer)	(SymName,Mode)	(i,i)
		(i,o)

Mode: 0=text 1=binary

filepos(file,real,int)	(SymName,Pos,Mode)	(i,i,i)
		(i,o,i)

Mode: 0=Start 1=Current 2=End

file__str(string,string)	(DosFile,Var)	(i,o)
		(i,i)
flush(file)	(SymName)	(i)
openappend(file,string)	(SymName,DosFile)	(i,i)
openmodify(file,string)	(SymName,DosFile)	(i,i)
openread(file,string)	(SymName,DosFile)	(i,i)

Built-in Predicate Declarations	Arguments	Flow Patterns
openwrite(file,string)	(SymName,DosFile)	(i,i)
readdevice(file)	(SymName)	(i) (o)
renamefile(string,string)	(OldName,NewName)	(i,i)
writedevic(file)	(SymName)	(i) (o)

Flow Control

Built-in Predicate Declarations	Arguments	Flow Patterns
bound(term)	(Term)	(i o)
cutbacktrack(integer)	(BtrackTop)	(i)
fail		()
free(term)	(Term)	(i o)
getbacktrack(integer)	(BtrackTop)	(o)
true		()

Input

Built-in Predicate Declarations	Arguments	Flow Patterns
inkey(char)	(Var)	(o)
keypressed		()
readchar(char)	(Var)	(o)
readint(integer)	(Var)	(o)
readln(string)	(Var)	(o)
readreal(real)	(Var)	(o)
readterm(term___ domain,term)	(Domain,Var)	(i,o)
unreadchar(char)	(Var)	(i)

Internal Database

Built-in Predicate Declarations	Arguments	Flow Patterns
assert(term)	(Term)	(i)
asserta(term)	(Term)	(i)
assertz(term)	(Term)	(i)
consult(string)	(DosFile)	(i)
consult(string,db__domain)	(DosFile,DbName)	(i,i)
retract(term)	(Term)	(i o)
retract(term,db__domain)	(Term,DbName)	(i o,i)
retractall(term)	(Term)	(i o)
retractall(term,db__domain)	(Term,DbName)	(i o,i)
save(string)	(DosFile)	(i)
save(string,db__domain)	(DosFile,DbName)	(i,i)

Machine Level

Built-in Predicate Declarations	Arguments	Flow Patterns
bios(int,regdom,regdom)	(Int,RegIn,RegOut)	(i,i,o)
RegIn = reg(AXi,BXi,CXi,DXi,Sli,Dli,DSi,ESi) RegOut = reg(AXo,BXo,CXo,DXo,Slo,Dlo,DSo,ESo)		
bios(int,regdom,regdom,int)	(Int,RegIn,RegOut,Flags)	(i,i,o,o)
bitand(int,int,int)	(X,Y,Z)	(i,i,o)
bitleft(int,int,int)	(X,Y,Z)	(i,i,o)
bitnot(integer,integer)	(X,Y)	(i,o)
bitor(int,int,int)	(X,Y,Z)	(i,i,o)
bitright(int,int,int)	(X,Y,Z)	(i,i,o)
bitxor(int,int,int)	(X,Y,Z)	(i,i,o)
debug		()
membyte(int,int,int)	(Segment,Offset,Byte)	(i,i,i)
		(i,i,o)

Built-in Predicate Declarations	Arguments	Flow Patterns
port_byte(integer, integer)	(PortNo, Value)	(i, i) (i, o)
ptr_dword(str, int, int)	(Pointer, Segment, Offset)	(i, o, o) (o, i, i)

Output

Built-in Predicate Declarations	Arguments	Flow Patterns
nl		()
write(term[, term...])	(Term[, Term...])	(i[, i...])
writeln(string, term[, term...])	(FormatStr, Term[, Term...])	(i, i[, i...])

Screen Handling

Built-in Predicate Declarations	Arguments	Flow Patterns
attribute(integer)	(Attr)	(i)
cursor(integer, integer)	(Row, Column)	(i, i) (o, o)
cursorform(integer, integer)	(Var, Var)	(i, i) (o, o)
field_attr(int, int, int, int)	(Row, Col, Len, Attr)	(i, i, i, i) (i, i, i, o)
field_str(int, int, int, str)	(Row, Col, Len, Var)	(i, i, i, i) (i, i, i, o)
scr_attr(int, int, int)	(Row, Col, Attr)	(i, i, i) (i, i, o)
scr_char(int, int, char)	(Row, Col, Var)	(i, i, i) (i, i, o)
scroll(integer, integer)	(Row, Col)	(i, i)
textmode(integer, integer)	(Height, Width)	(i, i) (o, o)

String Handling

Built-in Predicate Declarations	Arguments	Flow Patterns
concat(str,str,str)	(Front,Rest,Var)	(i,i,i) (i,i,o) (i,o,i) (o,i,i)
format(string,string, term[,term...])	(OutStr,FormatStr,Term [,Term...])	(i,i,i[,i...])
frontchar(str,char,str)	(Var,Front,Rest)	(i,i,i) (i,i,o) (i,o,i) (i,o,o) (o,i,i)
frontstr(int,str,str,str)	(StrLen,Var,Front,Rest)	(i,i,o,o)
fronttoken(str,str,str)	(Var,Token,Rest)	(i,i,i) (i,i,o) (i,o,i) (i,o,o) (o,i,i)
isname(string)	(Var)	(i)
str_len(string,integer)	(Var,Length)	(i,i) (i,o) (o,i)

System (DOS)

Built-in Predicate Declarations	Arguments	Flow Patterns
comline(string)	(Var)	(o)
dir(str,str,str)	(Path,FileSpec,DosFile)	(i,i,o)
dir(str,str,str,int,int,int)	(Path,Filespec,DosFile, Dir,Mask,ShowPath)	(i,i,o,i,i,i)

Dir: 0=no directories 1=directories

Mask: 0=no change 1=allow user to change mask

ShowPath: 0=don't show path 1=show path

Built-in Predicate Declarations	Arguments	Flow Patterns
envsymbol(string,string)	(EnvId,Symbol)	(i,o)
system(string)	(DosCommand)	(i)
system(str,int,int)	(DosCommand,Mode, RetCode)	(i,i,o)

Mode: 0=don't reset video 1=resetvideo

Windows

Built-in Predicate Declarations	Arguments	Flow Patterns
clearwindow		()
colorsetup(integer)	(Mode)	(i)
Mode: 0=window color 1=frame color		
existwindow(integer)	(No)	(i)
framewindow(integer)	(Attr)	(i)
framewindow(int, str,int,str)	(Attr,Name,Pos,Frame)	(i,i,i,i)
Pos: 255=Center Name <>255=Position		
gotowindow(integer)	(No)	(i)
makewindow(int,int,int, str,int,int,int,int)	(No,ScrAtt,FrameAtt, Name,Row,Col, Height,Width)	(i,i,i,i,i,i,i,i) (o,o,o,o,o,o,o,o)
makewindow(int,int,int, str,int,int,int,int, int,int,str)	(No,ScrAtt,FrameAtt, Name,Row,Col, Height,Width,Mode, NamePos,Frame)	(i,i,i,i,i,i,i,i,i,i) (o,o,o,o,o,o,o,o,o,o)

Mode: 0=Don't clear window 1=Clear window

NamePos: 255=Center Name <>255=Position

removewindow	()
--------------	-----

Built-in Predicate Declarations	Arguments	Flow Patterns
removewindow(integer, integer)	(No,Mode)	(i,i)
Mode: 0=don't refresh 1=refresh background		
resizewindow		()
resizewindow(int,int, int,int)	(Row,Height,Col,Width)	(i,i,i,i)
shiftwindow(integer)	(No)	(i) (o)
window__attr(integer)	(Attr)	(i)
window__str(string)	(Var)	(i) (o)

Miscellaneous

Built-in Predicate Declarations	Arguments	Flow Patterns
=		
beep		()
date(int,int,int)	(Year,Month,Day)	(i,i,i) (o,o,o)
findall(var,predicate,var*)	(Var,Pred,List)	(o,i o,o)
not(term)	(Term)	(i)
random(real)	(Var)	(o)
random(integer,integer)	(Max,Var)	(i,o)
snowcheck(symbol)	(Mode)	(i) (o)
Mode = on off		
storage(real,real,real)	(Stack,Heap,Trail)	(o,o,o)
sound(integer,integer)	(Duration,Frequency)	(i,i)
trace(symbol)	(Mode)	(i) (o)
Mode = on off		
time(int,int,int,int)	(Hours,Min,Sec,Hund)	(i,i,i,i) (o,o,o,o)

Functions

Built-in Predicate Declarations	Arguments	Flow Patterns
abs(real)	(Var)	(i)
arctan(real)	(Radians)	(i)
cos(real)	(Radians)	(i)
exp(real)	(Var)	(i)
ln(real)	(Var)	(i)
log(real)	(Var)	(i)
round(real)	(Var)	(i)
sin(real)	(Radians)	(i)
sqrt(real)	(Var)	(i)
tan(real)	(Radians)	(i)
trunc(real)	(Var)	(i)

NOTE: Both trunc() and round() return values within the integer range.

Operators (infix)

Arithmetic: +, -, *, /, mod, div

Relational: >, <, =, >=, <=, <>, ><

Trademarks

Borland Graphics

Interface is a
trademark of

Borland International Inc.

IBM PC®

International Business Machines Corporation

MultiMate®

MultiMate International Corporation

SideKick®

Borland International Inc.

Turbo C®

Borland International Inc.

Turbo Pascal®

Borland International Inc.

Turbo Prolog

Toolbox™

Borland International Inc.

Turbo Prolog®

Borland International Inc.

WordStar®

MicroPro International Corporation

Index

`*`, 203
`[]`, 164, 165
`,` 194, 195
`\\`, 186
`" "`, 155, 173, 195
`=`, 148-149
`!`, 128, 135
`%`, 196, 197
`#`, 180
`?`, 203
`\`, 195
`' '`, 155
`/*`, 62
`—`, 173

A

`add__new__drawing()` predicate, 316
`add__to__list()` predicate, 320
`AND`, 84-85, 87
`append()` predicate, 181, 255
Arguments
 arity and, 77-78
 declaring types of, 79-80

Arguments, *continued*

described, 76
editing, 200-202
example of using, 65-66
in external databases, 228-230
initializing, 176-177
in internal databases, 214
mathematical, 144
output, 271-272
partially bound, 255-256
for predicates, 161-162
reference, 271-272
transferring, 274
type checking of, 63
used with predicates, 336
See also Objects; Terms

Arity, 77-78

Arrow keys

for positioning window borders, 31
for resizing windows, 18

ASCII

files, 40, 199, 200
integer values, 155

`assert()` predicate, 210, 215, 221
`asserta()` predicate, 210, 215, 221, 228, 267-268
`assertz()` predicate, 210, 215, 221, 228, 321
 AUTOEXEC.BAT file, 11, 203
 Auto-indent mode, 201
 Auxiliary Editor window
 copying to, with Xcopy (F7) key, 44
 editing with Xedit (F8) key, 44-45
 function of, 20
B
 B+ trees, 233-240
 creating, 234
 index for, 240
 leaf nodes (pages) in, 235
 opening and closing, 235-236
 searching for key in, 237
 structure, 235
 updating, 236-237
 using, 238-240
 Backtracking, 90-98
 bread crumbs, 91-94
 with deterministic predicates, 138-139
 during recursion, 189-191
 forcing, 271
 looping and, 223-224, 270
 mechanism, 129-130
 preventing, 267
 recursion and, 172, 174
 tracing during, 94-97
 using cut with, 134-136, 137
 beep() predicate, 306
 BGI extension, 285
 BGI.LIB file, 301-302
 BGI predicates, 286-289, 291-292, 294-297, 298, 299, 300

bgifont compiler directive, 301
 BGIOBJ.EXE file, 302
 Block functions, 48, 49
 WordStar-like, 50-51
 Borland Graphics Interface (BGI), 281-302
 adding, to programs, 300-302
 built-in predicates for, 337-339
 colors and graphics in, 290-295
 described, 282
 domain declaration with, 336
 drawing in, 295-297
 graphics cards in, 283-284
 initializing Graphics mode in, 284-289
 text versus graphics in, 282-283
 viewports in, 289-290
 writing text in Graphics mode in, 297-300
 bt_close() predicate, 235-236
 bt_create() predicate, 234-235
 bt_open() predicate, 235-236

C

calc_interval_number() predicate, 322
 calc_number_interval() predicate, 322
 calculate_intervals() predicate, 322
 CALL goal port, 95, 98, 244, 246, 252
 Case
 for anonymous variables, 83
 changing, 51-52
 for constants, 76
 converting, 157-158

Case, continued

- for objects and relationships, 57
 - for predicates, 103, 314
 - structure, 127-128, 131-132
 - for symbols, 110
 - for variables, 72, 76
- cb_open() predicate, 227-228
- CH18EXE08.PRO file, 307
- chain_inserta() predicate, 228-229, 230
- chain_insertafter() predicate, 228-229
- chain_insertz() predicate, 228-229, 230
- chain_terms() predicate, 230, 231
- Characters, 112-113
- comparing, 155-156
 - converted to integers, 157
 - special formatting, 195, 197
- check_determ compiler directive, 266
- CHR extension, 298, 301
- Clauses
- causing, to fail, 133
 - described, 62-64, 87
 - disjunction with, 215-216
 - example of using, 64-66
 - versus procedural statements, 140
- Clauses section, 102, 123
- database as, 207
 - example of, 313, 314
 - with include compiler directive, 253
- client() predicate, 129-130, 138
- Clocksin, W. F., 6, 59, 64
- closefile() predicate, 187
- closegraph() predicate, 289
- Code
- array, 274

Code, continued

- for example program, 313-314
 - optimizing, 134
 - storing, 274
 - techniques for, 261-280
- Color
- choosing attributes of, 104-105
 - and graphics, 290-295
 - selecting, 30, 294-295
- Color Graphics Adapter (CGA)
- card, 282, 283-284, 286
 - color and, 290-292, 294
- Commands, 16, 21
- Comments
- adding, 61-62
 - describing predicate relationships, 77, 79
- Compile command (F9 key), 16, 66-67
- Compile menu, 16, 26-27
- Compiler
- directives, 244, 245
 - features, 243-261
- Compiler Directives section, 123, 277
- Compiling, 66-67, 261
- concat() predicate, 181, 198
- Condition
- terminating, 168-169, 175-177
- CONFIG.SYS file, 10
- Configuration
- colors, 30
 - described, 29
 - help lines, 36
 - keyboard, 34-36
 - Load, 29
 - miscellaneous settings for, 33
 - Save, 29
 - screen mode, 34

Configuration, *continued*

- Setup menu for, 29-36
- window size, 31

Constants section, 76, 117-118, 123, 313

consult() predicate, 216-217, 221

consult_db() predicate, 315

Control

- predicates, 128-132
- searching, 125, 132-139
- structures, 126-128

See also Flow control

Conversion, 339

Copyright notice/version

- number, 11, 14-15

count_intervals() predicate, 324

count_performance() predicate, 322-323

Cursor, 44-46

Cut, 134-137

- creating deterministic predicates with, 138-139
- with elements, 178
- removing, with recursion, 189-190

cut() predicate, 128, 271

D

Data

- objects, 77
- reading in, 191-193
- writing out, 193-196

Database section, 117, 119, 210, 221, 223

- example of, 313, 314

Databases

- adding data to, 316
- B+ trees and, 233-240
- described, 87, 207
- displaying files in, 316
- dynamic, 209
- facts, 267

Databases, *continued*

- global, 257
- as knowledge bases, 208-209
- local, 257
- naming, 316
- predicates, 267
- protecting, 315
- reference number, 229, 233
- static data in, 208
- storing values in, 205, 223-225

Databases, external, 225-233

- B+ trees and, 233-240
- B+ trees in, 342-343
- built-in predicates for, 341-343
- chains, 225, 228-230, 238-240, 341-342
- creating, 226-227
- handling, 231-232
- locations of, 226-227
- miscellaneous predicates for, 233
- opening and closing, 227-228
- retrieving data from, 230-231
- terms, 225, 228-230
- updating terms in, 232-233

Databases, internal, 209-220

- asserting and accessing facts, 210-216
- built-in predicates for, 345
- declarations, 210
- deleting facts in, 217-219
- multiple, 220-223
- saving and consulting, 216-217
- saving facts in, 219-220

Databases, multiple internal

- declaring, 221-223
- domains, 223

- Databases, multiple,
 - internal, *continued*
 - manipulating, 221
- DATAFILE.DBA file, 217
- db_bday() predicate, 232
- db_close() predicate, 228
- db_create() predicate, 226, 228, 229
- db_selector domain, 226
- dbasedom domain, 220, 223
- Debug window, 279
- debug_prj() predicate, 279-280
- Debugging
 - with compiler directives,
 - 244, 245, 250-253
 - executable programs,
 - 279-280
 - modules, 277-278
 - projects, 277-280
 - syntax, 68
 - tracing for, 98
- Declarations, 336
- deletefile() predicate, 202
- Deleting, 46-47
- detectgraph() predicate, 287-289
- determ keyword, 267
- diagnostics compiler directive,
 - 250
- Dialog window
 - function of, 19
 - Goal: prompt in, 69, 70, 71,
 - 81-82
 - leaving, 71
 - location of, 16
- dir() predicate, 202-203, 316
- Directories
 - preparing, 9-10
 - submenu, 32
- Directory window, 23-24
- Disks
 - kinds of, 8
 - preparing, 8-9
- Display mode, 201
- display() predicate, 198-199
- Do-While structure, 128
- Domains
 - database, 223
 - described, 62-64
 - example of using, 64-66
 - for lists, 163
 - standard, 110-113, 313, 336
 - user-defined, 110, 113-114,
 - 313-314
- Domains section, 108-114, 123
 - example of, 313, 314
 - external databases and, 226
 - with include compiler directive, 253
 - information types for, 109
 - standard domains for, 110-113
 - type checking in, 109-110
 - user-defined domains for,
 - 113-114
- DOS_FILE.DAT file, 227
- drawing() predicate, 317
- drawing_stats() predicate, 320-321, 322
- drawing_stats_facts() predicate,
 - 319
- E**
- Edit menu, 16, 26
- edit() predicate, 199-202
- Edit PRJ File selection, 256
- Edit screen, 17
- editmsg() predicate, 199
- Editor
 - block functions in, 48, 49,
 - 50-51
 - built-in predicates for, 340
 - changing case in, 51-52
 - cursor movement commands
 - for, 44-46

Editor, *continued*

- deleting text in, 46-47
- described, 13-14, 20, 39-40
- function keys for, 43-44, 45
- going to main menu from, 44, 51-52
- help in, 42-43
- hot keys for, 37, 52
- indent mode in, 48
- insert mode in, 47
- inserting text in, 46-47
- leaving, 41
- miscellaneous commands in, 51-52
- overwrite mode in, 47
- return status of, 202, 205
- saving in, 42
- searching and replacing in, 48, 50
- selecting, 40-41
- using, 199-202, 205

Editor window

- activating, 40-41
- changing status with F6 key, 44
- function of, 20
- Help (F1) key, 20
- initial display, 14-15
- location of, 16
- menu, 26
- for resizing or moving, 17-18
- running file in, 26
- tracing and, 251
- zooming with F5 key, 44
- See also* Auxiliary Editor window

Enhanced Graphics Adapter (EGA)

- card, 282, 283-284, 286
- color and, 290, 292-293, 295

Environment, 13-14

`eof()` predicate, 187-188

Error

- built-in predicates for controlling, 340-341
- preventing, in internal databases, 216-217
- trapping, 135-136, 189

ESC key, 21-36

- active window status and, 17-18
- function of, 21-22
- to leave editor, 41

EXE file extension, 26-27, 32, 203, 254, 261

`existfile()` predicate, 202

`exit` command, 24

Exiting, 12, 16, 24

Expert systems, 3, 5-6

Expressions, 147-150

F

Facts

- anonymous variables representing, 83
- checking, 70
- described, 59-61, 76
- listed in Clauses section, 63

FAIL goal port, 97, 98, 244

`fail()` predicate, 128, 129-130, 138, 223-224, 264, 271

`file_str()` predicate, 200, 205

`filepos()` predicate, 186-187

Files

- BGI object, 301-302
- built-in predicates for handling, 343-344
- compiling with F9 key, 44
- dividing, into modules, 253-261
- menu, 22-25
- object. *See* Object files
- opening, 185, 186

Files, *continued*

- predicates for handling, 185, 186-189, 202-203
- preparing autoexec, 10-11
- protecting, 315
- redirection in, 185-186
- source, 39-40
- working with, 183-189

Files menu

- Change dir command, 24
- described, 22-25
- Directory command, 24
- item, location of, 16
- Load command, 22-23
- menus from Pick option, 26-29
- New file command, 24
- Operating system command, 24-25
- Pick command, 24-25
- Quit command, 24
- Save command, 24
- Write to command, 24
- find__last__drawing__number() predicate, 316, 317
- findall() predicate, 179-180
- Flow control
 - built-in predicates for, 344
 - problems, 398
- Flow patterns
 - with global predicates, 255
 - with predicates, 336
- flush() predicate, 187
- Fonts
 - compiling text, 300-301
 - types of, 298-299
- for() predicate, 268-270
- For/Next loops, 128, 268-270
- format() predicate, 199, 300
- Formatting
 - characters, 195
 - specifiers, 196, 197

- frontchar() predicate, 198
- frontstr() predicate, 198
- fronttoken() predicate, 198, 263
- Function keys

- active window status and, 17-18
- defined in editor, 43-44, 45
- described, 16
- to leave editor, 41
- to save files in editor, 42

Functions

- built-in predicates for, 350
- described, 150-151
- radians or degrees with, 151-152
- rounding and truncating with, 152-153
- standard mathematical, 151
- Functors, 120

G

- get__age__code(), 138-139
- get__age__code() predicate, 131-132, 133-134, 135-136, 138
- get__answer() predicate, 314
- get__file() predicate, 205
- get__input() predicate, 180
- get__integer() goal, 192-193
- get__position() predicate, 205
- getbkcolor() predicate, 294
- getcolor() predicate, 295
- getpalette() predicate, 294
- Global calls, 278
- Global Database section, 123, 257
- Global Domains section, 123, 254
- Global Predicates section, 116-117, 123, 254
- Global section, 123
- GLOBALS.PRO file, 257, 260
- go() predicate, 129-131
- Goal ports, 244, 246-247

Goal section, 114-116, 123, 205, 279

example of, 314, 321-322, 325

main module and, 256

Goals

compound, 84-90

containing AND, 84-85

containing OR, 85-86

described, 62-64

example of using, 64-66

external, 64, 69, 70, 82

internal, 64

with multiple arguments, 78-80

questions and, 64, 69-70

rules used for new, 88-90

search process for, 72-73, 80, 88

tracing compound, 97-98

using not() with, 98-99

using variables in, 81-82

GRAPDECL.PRO file, 285-286, 287, 291-292, 296, 302

Graphics

cards, 283-284

and color, 290-295

drawing in, 295-297

drivers, 285, 301, 302

fonts, 298-299

hardware for, 8

systems, 285, 290

versus text, 282-283

viewports for, 289-290

Graphics mode, 283, 286

initializing, 284-290

writing text in, 297-300

H

Hardware

compatible, 7-8

for graphics, 8

Heap

array, 274

overflow, 274-275

Help

F1 key for, 44

index in editor, 42-43

Hot keys

addition of, 14

described, 37

for editor, 37, 41, 42, 52

exiting with, 12, 16, 24

list of, 12, 37

for menu item selection, 21

I

I/O redirection, 184-186

IF statement, 86-87

If-Then structure, 127-128, 131-132

If-Then-Else structure, 127-128, 132

include compiler directive, 252-253, 257

Indent mode (auto-indenting), 48, 201

Inference engine, 4, 6, 128

Information line, 16

initgraph() predicate, 285, 286-287, 301

Input, 183-205

built-in predicates for, 344
streams, 184-186, 191

Insert mode, 47

Inserting, 46-47

install command, 10

installh command, 10

Installing, 9-11

by unpacking batch files, 7-8, 14

disk manipulation in, 8-9

paths and autoexecs for, 10-11

- Installing, *continued*
 - subdirectories for, 9-10, 14
 - the Toolbox, 11
- Instantiating, 166-168
 - with free variables, 81
 - in internal databases, 214
 - in recursion, 175
 - to value of clause, 72-73
- Integers
 - converting, 157
 - described, 112, 143
 - hex notation and, 158
 - length of list as, 175
 - lists of, 163, 164
 - mod and div with, 145-146
 - with random numbering, 153-154
 - rounding to value of, 152
- interval() predicate, 324
- isname() predicate, 198
- K**
 - key_delete() predicate, 236-237
 - key_insert() predicate, 236-237
 - key_next() predicate, 237
 - key_prev() predicate, 237
 - key_search() predicate, 237
- Keys
 - reading, 306-309
 - See also* Function keys; Hot keys; *specific key names*
- L**
 - Language
 - declarative vs. procedural, 55-56
 - defining facts in, 59-61
 - objects and relationships of, 56-58, 59-60
 - procedural, 55-56, 140
 - syntax of, 58-59
 - last_num() predicate, 317
 - line() predicate, 296
 - linerel() predicate, 296
 - lineto() predicate, 296
 - LIST.PRO file, 260
 - list_length() predicate, 175-176
 - LIST_PRD.PRO file, 253
 - Lists, 122-123
 - adding elements to, 178
 - assigning values in, 174-176
 - building, dynamically, 179-180
 - compared with compound objects, 163
 - complex, 272-273
 - creating, 176-178, 181
 - data structure of, 163-166
 - declaring, 163
 - elements of, 164, 165, 172-174
 - empty (null), 165, 174, 176, 177
 - head/tail of, 165-166, 171, 178
 - initializing, 177
 - notation for, 164-166
 - recursion in, 168-174
 - singleton, 165
 - splicing together, 181
 - to do, 203-205
 - unifying, 166-168
 - using append() predicate for, 181
 - LITTF.OBJ file, 302
 - Load command (F3 key), 16, 44
 - Log Status menu, 251-252
 - Logic, 3, 5-6
 - Loops, 126-127
 - backtracking in, 191, 223-224, 270
 - conditional, 126, 129-130
 - counter, 126-127, 132, 261-264

Loops, *continued*

- For/Next, 268-270
- iterative, 262, 264-266
- nested, 269-270
- nondeterministic predicates
 - and, 138-139
- recursive, 262-264
- Repeat/Fail, 192, 264, 268, 275
- Repeat/Failing-Condition, 264-266
- tail-recursive, 190-191

LOTTERY.PRO file

- bringing in data in, 315-317
- calculating intervals in, 317-324
- described, 311-312
- getting started with, 312-313
- looking at code in, 313-314
- supporting predicates in, 314-315
- writing out results in, 324-333

lotto_stats() predicate, 316, 321-322

M

- Machine level, 345-346
- Main menu, 44, 51-52
- make_list() predicate, 179-180, 263
- makewindow() predicate, 290
- Mathematics, 143-158
 - adding random flare in, 153-155
 - comparing terms of, 155-158
 - evaluating expressions in, 147-150
 - functions of, 150-153
 - numbers as symbols in, 144-147
- Mellish, C. S., 6, 59, 64

member() predicate, 169-174, 178, 319-320, 324

Memory

- array sizes, 276-277
- arrays, 274
- management, 274-277

memory() predicate, 275-276

Menu

- bar, 14, 21
- creating, 136
- items. *See also* Files, Edit
 - Run, Compile, Options,
 - Setup menu items
- pull-down, 21

menu() predicate, 304-305

Message window, 16, 19

Modular programming

- compiling project in, 261
- defining project module in, 256
- described, 253-254
- global databases in, 257
- global include files in, 257-260
- global predicates and domains in, 254-255
- partially bound arguments in, 255-256
- writing program modules in, 256-257

MultiMate, 40

MY_PROJ.PRJ file, 260

MY_PROJ.PRO file, 257-258

N

- Natural-language processing, 5
- Negation, 271
- next() predicate, 270
- nondeterm keyword, 267-268
- Not() predicate, 98-99
- Notation
 - head/tail, 165-166, 171

Notation, *continued*

- hex, 158
- infix, 144-145, 146, 148, 149
- list, 164-166
- postfix, 144
- prefix, 144, 150

Numbers

- absolute values of, 150-151
- calculating power of, 154-155
- converting real, to integer values, 152-153
- as integers. *See* Integers
- random, 153-154
- real, 143, 157
- as symbols, 144-147

O

OBJ file extension, 26-27, 32

Object files, 39-40, 61

- BGI, 301-302
- compiling, 254

Objects

- comparing compound, 156-158
- compound, 119-120, 162, 163
- recursive data, 122
- and relationships, 56-58, 59-60
- as variables, 72
- See also* Arguments
- open_for_writing goal, 185
- openappend(), 185
- openappend() predicate, 325
- openmodify(), 185
- openread() predicate, 185, 187-188
- openwrite() predicate, 185

Operators

- arithmetic, 145, 146, 147
- built-in predicates for, 350
- comparison, 149

Operators, *continued*

- mod and div, 145-147
- order of evaluation of, 145
- position of, 144-145
- relational, 148, 155-156

Options menu, 28, 256

- item, location of, 16
- setting memory array sizes with, 276-277

OR statement, 85-86, 87, 214-215

ORDER.PRO file, 258-260

Output, 183-205

- built-in predicates for, 346
- streams, 184-186, 194

output_tallies() predicate, 325

outtext() predicate, 300

outtextxy() predicate, 300

Overwrite/Insert mode, 47, 201

P

part() predicate, 222-223

Paths, 10-11

Predicates

- arguments for, 161-162
- arity of, 77-78
- for calculating exponents, 154-155
- control, 128-132
- database, 267
- declarations, 102-103
- defining single, 131
- described, 62-64, 76
- diagnostics for user-defined, 250
- editing, 199-202
- effect of cut (!) on, 136-137
- equality, 148-149
- example of using, 64-66
- file-handling, 183, 185, 186-189, 202-205
- flow patterns of, 107-108

Predicates, *continued*

- global, 117, 254-256, 267
- local, 117
- look-up, 232
- matching, 174
- mathematical, 144
- miscellaneous, for external databases, 233
- modeling, 154-155
- nondeterministic, 138-139
- recursive, 169-171
- redirection, 184
- section, 102-108
- simulation, 278
- spying on, 246, 247-248
- string-handling, 191-202, 196-202
- supporting, 313-314
- system-defined, 104-107
- tail-recursive, 189-191, 246
- user-defined, 102-103
- See also* Predicates, built in; Relationships; *specific names*

Predicates, built-in, 335-350

- abbreviations in, 336
- for BGI, 286-289, 291-292, 294-297, 298, 299, 300, 337-339
- for comparing compound objects, 156-157
- for conversion, 339
- for data conversion, 157-158
- for editor, 340
- for error control, 340-341
- for external database B+ trees, 342-343
- for external database Chains, 341-342
- for file handling, 343-344
- for flow control, 344
- for functions, 350
- for input, 344

Predicates, built-in, *continued*

- for internal database, 345
- for machine level, 345-346
- miscellaneous, 349
- notation in, 336
- operators for, 350
- for output, 346
- for random numbers, 153-154
- for screen handling, 346
- for string handling, 347
- for system (DOS), 347-348
- for windows, 348-349

• Predicates section, 123

- example of, 313, 314
- with include compiler directive, 253

print_intervals() predicate, 326

print_report() predicate, 326

Printer Status window, 193

Printing

- example results, 324, 326, 327
- in Graphics mode, 300
- supplying values for, 196, 199
- trace results, 193, 252

Problem reduction, 168-169

Procedure

- described, 131
- statements of, 140

process_choice() clauses, 265-266

Program section(s)

- advanced, 116-119
- advanced declarations for, 119-123
- Clauses, 102
- Domains, 108-114
- Goal, 114-116
- list of, 123
- Predicates, 102-108

Programming in Prolog (Clocksin and Mellish), 6, 59, 64

Program(s)

- adding BGI to, 300-302
- adding rules to, 86-87
- checking facts of, 70
- compiling, 66-67, 253-261
- debugging, 68
- divisions of, 62-64
- example of, 64-73
- goals and questions of, 69-70
- processing, 274
- running, 69
- saving, 66, 79
- stand-alone, 114
- stopping, 71
- streams, 184-186
- syntax checking, 67-68

project compiler directive, 257

Prolog

- building blocks of, 76-80
- as declarative language, 55-56
- defining facts in, 59-61
- expert systems in, 4-5
- history of, 3
- logic of, 3, 5-6
- matching mechanism of, 80-84
- natural-language processing in, 5
- objects and relationships of, 56-58, 59-60
- relationship of data to, 87
- resolution and unification of, 6
- standards, 7
- starting, 11-12
- strengths of, 4-7
- syntax of, 58-59

prolog command, 11, 14

PROLOG.LOG file, 251-252

Q

Quit command, 12, 24

R

- random() predicate, 153-154
- read_file() predicate, 189-191
- readchar() predicate, 191-192
- readdevice() predicate, 184
- readin() predicate, 191
- readint() predicate, 191-192
- readkey() predicate, 306-309
- README.COM file, 9, 188
- readreal() predicate, 191-192
- Real data type, 112
- Recursion
 - creating lists with, 176-177
 - definitions of, 118-119
 - described, 168
 - in list processing, 169-170
 - problem reduction with, 168-169
 - tail, 189-191, 246
 - winding down into, 172-174
- REDO goal port, 93, 96, 98, 244
- Relationships
 - defining, 161-162
 - and objects, 56-58, 59-60
 - See also* Predicates
- remove_dups() predicate, 177-178
- rename_old_database() predicate, 314-315
- repeat() predicate
 - backtracking and, 264, 268
 - creating menu selection routine with, 265-266
 - exempting flagging from, 267
- Repeat-Until statement, 128
- Repeat/Fail loop, 192, 264, 266, 268, 275

Repeat/Failing-Condition loop,
264-266
 report__2__printer() predicate,
327
 report__intervals() predicate, 325
 Resolution principle, 6
 RESULTS.DAT file, 325, 327
 RESULTS.DBA file, 324
 retract() predicate, 217-219, 221,
336
 retract__drawings() predicate, 318
 retractall() predicate, 217, 219,
221, 318-319
 RETURN goal port, 96, 98, 244,
246, 252
 reverse() predicate, 252
 Robinson, J. Alan, 6
 round() function, 152-153
 Rules
 in Clauses section, 63
 described, 86-87
 subgoals and, 133
 using, 88-90
 Run menu, 16, 26
 run() predicate, 190, 215
 Run screen, 17
 Running, 7-9, 69
 program, 69

S

save() predicate, 216-217, 221
 save__position() predicate, 205
 Saving

 with ALT-F2 hot keys, 66
 with F2 key, 16, 44, 66
 facts in internal databases,
219-220
 internal database, 216-217
 programs, 66, 79

Screen

 built-in predicates for han-
dling, 346

Screen, *continued*

 resolution, 283, 284
 selecting colors for, 30

Searching

 backward-chaining, 140-141
 controlling, 125
 for information, 71-72
 reducing time for, 132-139
 and replacing, 44, 48, 50
 strategies, 140-141
 using cut with, 134-136
 using matching mechanism
 for, 133-134

setallpalette() predicate, 294

setbkcolor() predicate, 291-292,
294

setcolor() predicate, 294-295

setgraphbufsize() predicate, 297

setlinestyle() predicate, 296

setpalette() predicate, 294

settextstyle() predicate, 298-299,
300-301

Settings, 21

Setup menu, 29

 for colors, 30
 for directories, 32
 for help lines, 36
 item, location of, 16
 for keyboard configuration,
34-36
 for miscellaneous settings, 33
 Save configuration com-
mand, 20
 for screen mode, 34
 for window size, 31

setusercharsize() predicate, 299

setviewport() predicate, 289-290

SHIFT key, 31

shorttrace compiler directive,
246-247

SideKick, 40

- SideKick, *continued*
 - commands styled like, 45
 - Notepad, 20
- Sound, 305-306
- sound() predicate, 306
- Source code modules, 116-117
- Stack
 - array, 274
 - decreasing size of, 254
 - overflow, 189-191, 274
- Starting, 11-12, 14-16
- Statements
 - assignment, 149
 - procedural, 140
- Stopping, 71
- storage() predicate, 275
- str_char() predicate, 157
- str_int() predicate, 157
- str_len() predicate, 196-197
- str_real() predicate, 157
- Strings, 111
 - built-in predicates for handling, 347
 - comparing, 155, 156
 - converting, 157
 - dissecting, 198
 - format, 196
 - working with, 191-202
- Switch command (F6 key), 16, 128
- SymbolicFileName, 184-185, 186, 187, 227
- Symbols, 110-111, 155
- Syntax
 - checking, 67-68
 - creating programs and, 311
 - described, 58-59
 - in internal databases, 216-217
- SYS file extension, 29
- T**
 - tally_numbers() predicate, 319, 321, 325
 - tally_y() predicate, 319-320, 321
 - TDOMS.PRO file, 303-304
 - term_delete() predicate, 232-233
 - term_replace() predicate, 232-233
 - Terminating condition, 168-169, 175, 177
 - Terms, 76-77
 - TEST.DAT file, 200
 - Text
 - compiling fonts for, 300-301
 - outputting, 300
 - versus graphics, 282-283
 - writing, in Graphics mode, 297-300
 - Text mode, 282
 - TO_DO.LST file, 205
 - Toggles, 21
 - Toolbox, 281, 302-305
 - described, 302-303
 - graphics and, 8
 - installing, 11
 - menus in, 303-305
 - parsing with, 5
 - user interface (UI) section of, 303
 - windows and, 17
 - TPREDS.PRO file, 303-304
 - trace compiler directive, 246-247
 - Trace mode, 244, 246-247, 251-252
 - trace() predicate, 248-249, 251
 - Trace Status menu, 251-252
 - Trace Status window, 193
 - Trace window
 - free variables as shown on, 173
 - function of, 18-19
 - location of, 16
 - recursion and, 176
 - for resizing or moving, 17-18
 - simulating, 279
 - tracing and, 251

Tracing

- as compiler directive, 244
- with compiler directives, 250-253
- compound goal, 97-98
- during backtracking, 94-97
- goal ports in, 244
- outputting results of, 193
- recursive calls, 193
- recursive predicates, 173-174, 175
- sections of code, 248-249

Trail array, 274

true() predicate, 128, 215, 321

trunc() function, 152-153

Turtle Graphics, 303

Type

- checking, 109-110
- error, 114
- information, 109
- multiple domain, 121-122
- recursive data, 122

U

Unification, 6, 166-168

- testing with, 171-172

unpack command, 8

UNPACK program, 8

upper_lower() predicate, 157

V

Values

- assigning, in recursion, 174-176
- incrementing, 149-150
- returning, from deleted facts, 218-219
- using database to store, 223-225

Variables

- anonymous (blank), 83-84, 89, 170, 192

Variables, *continued*

- binding values to, 76, 82, 90
- described, 72-73, 76

Variables, free, 81-82, 90

- binding values to, 166
- and equality predicate, 149
- in internal databases, 214
- and mathematical expressions, 147
- with not(), 99
- with random numbers, 153-154
- as shown on Trace window, 173

Variables

- freeing, through backtracking, 93-94, 150
- keeping track of reference, 274
- in rules, 89-90
- unification and, 166-168

Viewports, 289-290

W

Wildcards, 23, 24

Windows, 16-20

- built-in predicates for, 348-349
- changing status with F6 key, 44
- choosing attributes for, 104-105
- described, 16-17
- displaying, 14-15
- making and writing to, 104-106
- moving, 17-18
- saving configurations of, 20
- selecting colors for, 30
- selecting size and position of, 17-18, 31

Windows, *continued*

See also Editor, Dialog, Message, Trace windows

word_count() predicate, 263-264

Word-wrap mode, 202

WordStar, 40

 commands styled like, 45

 standard, 20

WORK.PRO file, 12, 14-16,

66

write() predicate, 190, 194-195,

300

write_tallies() predicate, 325

writedvice() predicate, 184, 194

writef() predicate, 194, 196, 199,

300

The manuscript for this book was prepared and submitted to Osborne/McGraw-Hill in electronic form. The acquisitions editor for this project was Cynthia Hudson, the technical reviewer was Keith Hawes, and the project editor was Lindy Clinton.

Text design by Judy Wohlfrom, using Baskerville for text body and Megaron for display.

Cover art by Bay Graphics Design Associates. Color separation by Colour Image. Cover supplier, Phoenix Color Corporation. Screens produced with InSet, from InSet Systems, Inc. Book printed and bound by R.R. Donnelley & Sons Company, Crawfordsville, Indiana.

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Here's More Help From the Experts

Now that you've developed even better computer skills with *Using Turbo Prolog™, Second Edition*, let us suggest the following related titles that will help you use your computer to greater advantage.

Advanced Turbo Prolog™ Version 1.1

by Herbert Schildt

Herb Schildt now applies his expertise to Borland's remarkable Turbo Prolog™ language development system, specifically designed for fifth-generation language programming and the creation of artificial intelligence on your IBM® PC. *Advanced Turbo Prolog™* has been extensively revised to include Turbo Prolog version 1.1. The new Turbo Prolog Toolbox™, which offers more than 80 tools and 8,000 lines of source code, is also described in detail. Schildt focuses on helping you progress from intermediate to advanced techniques by considering typical AI problems and their solutions. Numerous sample programs and graphics are used throughout the text to sharpen your skills and enhance your understanding of the central issues involved in AI. Expert systems, problem solving, natural language processing, vision and pattern recognition, robotics, and logic are some of the applications that Schildt explains as he leads you to Turbo Prolog mastery.

\$21.95 p

0-07-881285-2, 350 pp., 7³/₈ x 9¹/₄

*The Borland-Osborne/McGraw-Hill
Programming Series*

Using Turbo Pascal®

by Steve Wood

Using Turbo Pascal® gives you a head start with Borland's acclaimed compiler, which has become a worldwide standard. Programmer Steve Wood has completely rewritten the text and now provides programming examples that run under MS-DOS®, as well as new information on memory resident applications, in-line code, interrupts, and DOS functions. If you're already programming in Pascal or any other high-level language, you'll be able to write programs that are more efficient than ever. *Using Turbo Pascal®* discusses program design and Pascal's syntax requirements, and thoroughly explores Turbo Pascal's features. Then Wood develops useful applications and gives you an overview of some of the advanced utilities and features available with Turbo Pascal. *Using Turbo Pascal®* gives you the skills to become a productive programmer—and when you're ready for more, you're ready for *Advanced Turbo Pascal®*.

\$19.95 p

0-07-881284-4, 350 pp., 7³/₈ x 9¹/₄

*The Borland-Osborne/McGraw-Hill
Programming Series*

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Using Turbo Pascal® Version 4

by Steve Wood

Using Turbo Pascal®, the book that gives you a head start with Borland's internationally acclaimed compiler, now appears in a special edition that covers the new Turbo Pascal® version 4. This version offers significantly faster compilation speed, separate compilation of units, project management facilities, and the Borland Graphics Interface. Author Steve Wood provides programming examples that run under MS-DOS®, as well as information on memory-resident applications, in-line codes, interrupts, and DOS functions. If you're already programming in Pascal or any other high-level language, you'll be able to write programs that are more efficient than ever. *Using Turbo Pascal®, Version 4* discusses program design and Pascal's syntax requirements, and thoroughly explores Turbo Pascal's features. Wood also develops useful applications and gives you an overview of some of the advanced utilities and features available with Turbo Pascal version 4. *Using Turbo Pascal®, Version 4* helps you develop the skills to become a productive programmer—and when you're ready for more, you're ready for *Advanced Turbo Pascal® Version 4*.

\$19.95p

0-07-881356-5, 500 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill
Programming Series

Advanced Turbo Pascal

by Herbert Schildt

Advanced Turbo Pascal® is the book you need to learn superior programming skills for the leading Pascal language development system. Revised and expanded, *Advanced Turbo Pascal®* now covers Borland's newly released Turbo Pascal Database Toolbox®, which speeds up database searching and sorting, and the Turbo Pascal Graphix Toolbox®, which lets you easily create high-resolution graphics. And, *Advanced Turbo Pascal®* includes techniques for converting Turbo Pascal for use with Borland's hot new compiler, Turbo C®. Schildt provides many programming tips to take you on your way to high performance with Turbo Pascal. You'll refine your skills with techniques for sorting and searching; stacks, queues, linked lists, and binary trees; dynamic allocations; expression parsing; simulation; interfacing to assembly language routines; and efficiency, porting, and debugging. For instruction and reference, *Advanced Turbo Pascal®* is the best single resource for serious programmers.

\$21.95p

0-07-881283-6, 350 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill
Programming Series

Advanced Turbo Pascal® Version 4

by Herbert Schildt

This separate edition of *Advanced Turbo Pascal®* is devoted exclusively to Borland's newly released version 4 which features faster compilation speed. Schildt provides many programming tips to take you on your way to high performance programming with Turbo Pascal. You'll refine your skills with techniques for sorting and searching; stacks, queues, linked lists, and binary trees; dynamic allocation; expression parsing; simulation; interfacing to assembly language routines; and efficiency, porting, and debugging. *Advanced Turbo Pascal Version 4* also covers the Turbo Pascal Database Toolbox®, which speeds up database searching and sorting, and the Turbo Pascal Graphix Toolbox®, which lets you easily create high-resolution graphics. This is the best single resource for serious Turbo Pascal 4 programmers.

\$21.95p

0-07-881355-7, 370 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill
Programming Series

Turbo Pascal®: The Complete Reference

by Stephen O'Brien

Turbo Pascal®: The Complete Reference is an important addition to both the *Borland-Osborne/McGraw-Hill Programming Series* and *Osborne's Complete Reference Series*. Now programmer Stephen O'Brien has written the first single resource that provides both expert and novice programmers with the entire range of Turbo Pascal's techniques, all illustrated in short examples and applications. Every aspect of Turbo Pascal is thoroughly described, including topics that were previously unavailable in one reference, such as memory-resident programs, DOS and BIOS services, and assembly language routines. *Turbo Pascal®: The Complete Reference* is clear, comprehensive, and organized for quick fact-finding. An ideal desktop resource you can refer to whenever you're programming with Borland's renown Turbo Pascal compiler.

\$27.95p, Hardcover Edition

0-07-881350-6 640 pp., 7³/₈ x 9¹/₄

\$24.95p, Paperback Edition

0-07-881290-9, 750 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill
Programming Series

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Turbo Pascal® Version 4: The Pocket Reference

by Kris Jamsa

\$5.95 p

0-07-881379-4, 120 pp., 4 1/4 x 7

Turbo Pascal® Programmer's Library

by Kris Jamsa and Steven Nameroff

You can take full advantage of Borland's famous Turbo Pascal® with this outstanding collection of programming routines. Now revised to cover Borland's new Turbo Pascal Numerical Methods Toolbox™, the *Turbo Pascal® Programmer's Library* includes a whole new collection of routines for mathematical calculations. You'll also find new date and time routines. Kris Jamsa, author of *DOS: The Complete Reference* and *The C Library*, and Steven Nameroff give experienced Turbo Pascal users a varied library that includes utility routines for Pascal macros as well as routines for string and array manipulation, records, pointers, and pipes. You'll find I/O routines and a discussion of sorting that covers bubble, shell, and quick-sort algorithms. And there's even more... routines for the Turbo Pascal Toolbox® and the Turbo Pascal Graphics Toolbox® packages. It's all here to help you become the most effective Turbo Pascal programmer you can be.

\$21.95 p

0-07-881286-0, 625 pp., 7 3/8 x 9 1/4

*The Borland-Osborne/McGraw-Hill
Programming Series*

Turbo Pascal® Programmer's Library Version 4

by Kris Jamsa and Steven Nameroff

You can take full advantage of Borland's newest version of Turbo Pascal® with this outstanding collection of programming routines. Now this separate edition of *Turbo Pascal® Programmer's Library* is devoted to version 4 and the new version 4 toolboxes. Kris Jamsa, author of *DOS: The Complete Reference*, *DOS Power User's Guide*, and *The C Library*, and Steven Nameroff give experienced Turbo Pascal users a varied library that includes utility routines for Pascal macros as well as routines for string and array manipulation, records, pointers, and pipes. You'll find I/O routines and a discussion of sorting that covers bubble, shell, and quick-sort algorithms. And there's even more—routines for the Turbo Pascal 4 toolboxes including, Turbo Pascal Database Toolbox®, Turbo Pascal Graphics Toolbox®, and Turbo Pascal Numerical Methods Toolbox®.

\$22.95 p

0-07-881368-9, 600 pp., 7 3/8 x 9 1/4

*The Borland-Osborne/McGraw-Hill
Programming Series*

Turbo C®: The Complete Reference

by Herbert Schildt

Herb Schildt's *C: The Complete Reference* which has topped best-seller charts across the country, is now followed by a special edition for Borland's highly acclaimed Turbo C® compiler. In *Turbo C®: The Complete Reference*, programmers at every level of Turbo C expertise will find all commands, functions, codes, and applications listed and described. *Turbo C®: The Complete Reference* also includes coverage of Borland's new Turbo C version 1.5, including the graphics functions and the Turbo C Librarian. Schildt describes the elements in the Turbo C language and environment, considers algorithms and applications, and discusses software development in Turbo C. An informative appendix points out the differences between Turbo C and Kernigan and Richie's C compiler. Add another bonus—money-saving coupons for popular Turbo C add-on products from Borland.

\$27.95 p, Hardcover Edition

0-07-881373-5, 850 pp., 7 3/8 x 9 1/4

\$24.95 p, Paperback Edition

0-07-881346-8, 850 pp., 7 3/8 x 9 1/4

*The Borland-Osborne/McGraw-Hill
Programming Series*

Turbo C®: The Pocket Reference

by Herbert Schildt

\$5.95 p

0-07-881381-6, 120 pp., 4 1/4 x 7

Using Turbo C®

by Herbert Schildt

Here's the official book on Borland's tremendous new C compiler. *Using Turbo C®* is for all C programmers, from beginners to seasoned pros. Master programmer Herb Schildt devotes the first part of the book to helping you get started in Turbo C. If you've been programming in Turbo Pascal® or another language, this orientation will lead you right into Turbo C fundamentals. Schildt's emphasis on good programming structure will start you out designing programs for greater efficiency. With these basics, you'll move on to more advanced concepts such as pointers and dynamic allocation, compiler directives, unions, bit-fields, and enumerations, and you'll learn about Turbo C graphics. When you've finished *Using Turbo C®*, you'll be writing full-fledged programs that get professional results.

\$19.95 p

0-07-881279-8, 350 pp., 7 3/8 x 9 1/4

*The Borland-Osborne/McGraw-Hill
Programming Series*

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Advanced Turbo C®

by Herbert Schildt

Ready for power programming with Turbo C®? You'll find the expertise you need in *Advanced Turbo C®*, the Borland/Osborne book with the inside edge. In this instruction guide and lasting reference, Herb Schildt, the author of five acclaimed books on C, takes you the final step of the way to Turbo C mastery. Each stand-alone chapter presents a complete discussion of a Turbo C programming topic so you can pinpoint the information you need immediately.

Advanced Turbo C® thoroughly covers sorting and searching; stacks, queues, linked lists, and binary trees; operating system interfacing; statistics; encryption and compressed data formats; random numbers and simulations; and expression parsers. In addition, you'll learn about converting Turbo Pascal® to Turbo C and using Turbo C graphics. *Advanced Turbo C®* shows you how to put the amazing compilation speed of Turbo C into action on your programs.

\$22.95 p

0-07-881280-1, 325 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill Programming Series

Using Turbo BASIC®

by Frederick E. Mosher and David I. Schneider

Using Turbo BASIC® is your authoritative guide to Borland's incredible new compiler that offers faster compilation speeds than any other product on the market. *Using Turbo BASIC®* is packed with information for everyone from novices to seasoned programmers. Authors Mosher and Schneider, two accomplished programmers, introduce you to the Turbo BASIC® operating environment on the IBM® PC and PC-compatibles, and discuss the interactive editor and the BASIC language itself. You'll learn about recursion, math functions, graphics and sound functions, and conversions from IBM BASICA to Turbo BASIC.

\$19.95 p

0-07-881282-8, 350 pp., 7³/₈ x 9¹/₄

The Borland-Osborne/McGraw-Hill Programming Series

C: The Complete Reference

by Herbert Schildt

For all C programmers, here's an encyclopedia of C terms, functions, codes, and applications. Arranged for quick fact-finding, *C: The Complete Reference* includes sections covering C basics, C library functions by category, various algorithms and C applications, and the C programming environment. You'll also find coverage of C++, C's newest direction, as well as full information on the UNIX® C de facto standard and the new proposed ANSI standard. Includes money-saving coupons for C products.

\$27.95 p, Hardcover Edition

0-07-881313-1, 740 pp., 7³/₈ x 9¹/₄

\$24.95 p, Paperback Edition

0-07-881263-1, 740 pp., 7³/₈ x 9¹/₄

C: The Pocket Reference

by Herbert Schildt

Speed up your C programming with *C: The Pocket Reference*, written by master programmer Schildt, the author of twelve Osborne/McGraw-Hill books. This quick reference is packed with vital C commands, functions, and libraries. Arranged alphabetically for easy use.

\$5.95 p

0-07-881321-2, 120 pp., 4¹/₄ x 7

C Made Easy

by Herbert Schildt

With Osborne/McGraw-Hill's popular "Made Easy" format, you can learn C programming in no time. Start with the fundamentals and work through the text at your own speed. Schildt begins with general concepts, then introduces functions, libraries, and disk input/output, and finally advanced concepts affecting the C programming environment and UNIX™ operating system. Each chapter covers commands that you can learn to use immediately in the hands-on exercises that follow. If you already know BASIC, you'll find that Schildt's C equivalents will shorten your learning time. *C Made Easy* is a step-by-step tutorial for all beginning C programmers.

\$18.95 p

0-07-881178-3, 350 pp., 7³/₈ x 9¹/₄

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Advanced C, Second Edition

by Herbert Schildt

Experienced C programmers can become professional C programmers with Schildt's nuts-and-bolts guide to advanced programming techniques. Now thoroughly revised, *Advanced C, Second Edition* covers the new ANSI standard in addition to the Kernighan and Ritchie C used in the first edition. All the example code conforms to the ANSI standard. You'll find information you need on sorting and searching; queues, stacks, linked lists, and binary trees; dynamic allocation, interfacing to assembly language routines and the operating system; expression parsing; and more. When you finish reading *Advanced C*, you'll be ready for the slick programming tricks found in Schildt's twelfth book for Osborne, C: Power User's Guide.

\$21.95 p

0-07-881348-4, 353 pp., 7³/₁₆ x 9¹/₄

C: Power User's Guide

by Herbert Schildt

Make your C programs sizzle! All the bells, whistles, and slick tricks used to get professional results in commercial software are unveiled to serious programmers in *C: Power User's Guide*. In his eleventh book for Osborne/McGraw-Hill, Schildt shows you how to build a Borland type interface, develop a core for a database, create memory resident programs, and more. Schildt combines theory, background, and code in an even mix as he excites experienced C programmers with new features and approaches. Learn master techniques for handling menus, windows, graphics, and video game programming. If hashing is what you're after, it's here too, along with techniques for using the serial port and sorting disk files. OS/2™ level programming with specific OS/2 functions is also covered. Before you send your programs out to market, consult Schildt for the final touches that set professional software apart from the rest.

\$22.95 p

0-07-881307-7, 384 pp., 7³/₁₆ x 9¹/₄

The C Library

by Kris Jamsa

Design and implement more effective programs with the wealth of programming tools that are offered in *The C Library*. Experienced C programmers will find over 125 carefully structured routines ranging from macros to actual UNIX™ utilities. There are tools for string manipulation, pointers, input/output, array manipulation, recursion, sorting algorithms, and file manipulation. In addition, Jamsa provides several C routines that have previously been available only through expensive software packages. Build your skills by taking routines introduced in early chapters and using them to develop advanced programs covered later in the text.

\$19.95 p

0-07-881110-4, 220 pp., 7³/₁₆ x 9¹/₄

Artificial Intelligence Using C

by Herb Schildt

With Herb Schildt's newest book, you can add a powerful dimension to your C programs—artificial intelligence. Schildt, a programming expert and author of seven Osborne books, shows C programmers how to use AI techniques that have traditionally been implemented with Prolog and LISP. You'll utilize AI for vision, pattern recognition, robotics, machine learning, logic, problem solving, and natural language processing. Each chapter develops practical examples that can be used in the construction of artificial intelligence applications. If you are building expert systems in C, this book contains a complete expert system that can easily be adapted to your needs. Schildt provides valuable insights that allow even greater command of the systems you create.

\$21.95 p

0-07-881255-0, 432 pp., 7³/₁₆ x 9¹/₄

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Advanced Graphics in C: Programming and Techniques

by Nelson Johnson

Add graphics to your C programs and you'll add significant capabilities to your software. With *Advanced Graphics in C* you'll be able to write graphics programs for the IBM® EGA (Enhanced Graphics Adapter), the de facto standard for high-quality graphics programming on the IBM PC.

Advanced Graphics in C features a special, complete graphics program called GRAPHIQ, that provides a whole toolkit of all the routines you'll need for graphics operations, and even gives you code for a rotatable and scalable character set. Johnson shows you how to use GRAPHIQ to implement or adapt graphics in your C programs. GRAPHIQ is full of tools not available elsewhere.

Advanced Graphics in C also offers an entire stroke/front character set; code for the AT&T Image Capture Board; and information on serial and parallel interfacing to mice, light pens, and digitizers.

\$22.95p

0-07-881257-7, 670 pp., 7³/₁₆ x 9¹/₄

Using QuickC®

by Werner Feibel

Learn QuickC® programming with Microsoft's speedy new compiler. Because QuickC is compatible with Microsoft's new C compiler version 5.0, this book offers the extra benefit of teaching you to program in both environments. Feibel, a C instructor and programming consultant, leads beginning C programmers from fundamentals to intermediate-level techniques, while providing experienced C programmers with a lasting reference. After a brief overview, *Using QuickC®* covers syntax and data types, operators, and expressions. Then Feibel tackles functions, arrays and pointers, the preprocessor, structures, unions, and files. Finally, there is a discussion of QuickC and its affiliation with the operating system and the hardware. Feibel makes a special effort to show you QuickC's unique features. Numerous examples are used in the book so that you can gain a better understanding of concepts. You'll even solve application problems by writing your own program solutions.

\$19.95p

0-07-881292-5, 375 pp., 7³/₁₆ x 9¹/₄

Advanced QuickC®

by Werner Feibel

Using QuickC®, Feibel's first QuickC book, explains beginning-to-intermediate techniques. Now *Advanced QuickC®* takes up where *Using QuickC®* left off. Build complex QuickC® programs that perform a variety of tasks with Feibel's advanced guide to Microsoft's compiler. If you already know C programming, you'll be able to use Feibel's algorithms, data structures, and code to best advantage. After reviewing fundamentals such as recursion, linked lists, queues, stacks, and trees, Feibel discusses applications that use sorting and searching, expression parsing, and data massage as well as graphs and networks. Feibel also discusses a range of functions and programs for accomplishing more specialized tasks, such as communicating with the operating system, generating random numbers, performing statistical analyses, and simulation based on common distribution. Feibel's well-paced text clearly instructs you in the finer points of QuickC so you can write faultless, professional programs.

\$21.95p

0-07-881352-2, 400 pp., 7³/₁₆ x 9¹/₄

OS/2™: The Pocket Reference

by Kris Jamsa

Computer user's memory loss is a common ailment that plagues even the most competent OS/2™ aficionado. Now, Osborne/McGraw-Hill has a remedy that's guaranteed for immediate results—*OS/2™: The Pocket Reference*. In these pages, you'll find complete examples of every important OS/2 command and system configuration entry, each accompanied by easy-by-follow tips and explanations. All entries are listed alphabetically.

\$5.95p

0-07-881377-8, 128 pp., 4¹/₄ x 7

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Using OS/2™

by Kris Jamsa

Microsoft's new operating system is now described for beginners and experienced computer users alike. *Using OS/2™* quickly moves from fundamental to advanced techniques and covers major OS/2 strengths in detail, including multi-tasking. Jamsa, a programmer for the United States Air Force and the highly regarded author of the acclaimed *DOS: The Complete Reference* and other Osborne books, draws on his extensive programming background to present OS/2 concepts in a clear, concise manner. You'll learn how to install OS/2 and learn the use of basic and advanced commands. Then you'll be ready for important techniques—redirection of I/O, system configuration and multi-tasking. A complete OS/2 command summary is provided.

\$19.95p

0-07-881306-9, 600 pp., 7³/₈ x 9¹/₄

OS/2™ Programmer's Guide

by Ed Iacobucci

Foreword by Bill Gates

Learn OS/2™ from an expert! Ed Iacobucci, who lead the IBM® OS/2 design team, shows experienced programmers how to use version 1.0 of the new and powerful multi-tasking operating system built expressly for 80286 and 80386-based personal computers. Iacobucci gives you a complete overview of 80286 protected mode, including valuable tips and techniques which you can use to build your first protected mode programs. Iacobucci provides a complete overview of the OS/2 system, filled with insights which help you understand how and why the system works. Learn about dynamic linking and the system API; memory management in a protected environment; OS/2 multitasking; advanced inter-process communications facilities; the system I/O capabilities; session management, user interface, utilities, and more. You'll also learn how to use many of these concepts in practical programming examples that include using the I/O facilities, allocating and managing memory, using basic and advanced multitasking features, and building a "pop-up" application with keyboard monitors.

\$24.95p

0-07-881300-X, 650 pp., 7³/₈ x 9¹/₄

DOS: The Complete Reference

by Kris Jamsa

Jamsa has all the answers to all of your questions about DOS through version 3.X. Every PC-DOS and MS-DOS® user will find essential information, whether you need an overview of the disk operating system or a reference for advanced programming and disk management techniques. Each chapter begins with a discussion of specific applications followed by a list of commands used in each. All commands are presented in the same clear, concise format: description, syntax, discussion of arguments or options, and examples. Jamsa provides comprehensive coverage of advanced DOS commands, EDLIN, Microsoft® Windows™, DOS error messages, and customization of your system via CONFIG.SYS.

\$27.95p, Hardcover Edition

0-07-881314-X, 1042 pp., 7³/₈ x 9¹/₄

\$24.95p, Paperback Edition

0-07-881259-3, 1042 pp., 7³/₈ x 9¹/₄

DOS: The Pocket Reference

by Kris Jamsa

Picture yourself working away efficiently on the computer when all of a sudden ... that DOS command you need just slips out of your mind and into space. Here, for the first time ever, is a *Pocket Reference* for PC and MS-DOS® through version 3.3. In this handy guide, you'll find complete examples of every DOS command with easy-to-follow tips and explanations. All the commands are listed alphabetically.

\$5.95p

0-07-881376-X, 128 pp., 4¹/₄ x 7

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

DOS Made Easy

by Herbert Schildt

Previous computer experience is not necessary to understand this concise, well-organized introduction that's filled with short applications and exercises. Schildt walks you through all the basics, beginning with an overview of a computer system's components and a step-by-step account of how to run DOS for the first time. Once you've been through the initial setup, you'll edit text files, use the DOS directory structure, and create batch files. As you feel more comfortable with DOS, Schildt shows you how to configure a system, handle floppy disks and fixed disks, and make use of helpful troubleshooting methods. By the time you've gone this far, you'll be ready for total system management—using the printer, video modes, the serial and parallel ports, and more. *DOS Made Easy* takes the mystery out of the disk operating system and puts you in charge of your PC.

\$19.95 p

0-07-881295-X, 385 pp., 7³/₈ x 9¹/₄

The Osborne/McGraw-Hill MS-DOS® User's Guide

by Paul Hoffman and Tamara Nicoloff

A comprehensive guide to the MS DOS® operating system, this book is designed to familiarize you with all the versions of this powerful system from Microsoft. Ideal for beginners and experienced users alike, this guide covers each computer running MS DOS®, gives the version it runs and any improvements the manufacturer has made to the system. It also gives complete information on the PC DOS version designed for the IBM® PC. Additional programs and reference material make this guide a tool of lasting value.

\$18.95 p

0-07-881131-7, 250 pp., 7¹/₂ x 9¹/₄

DOS: Power User's Guide

by Kris Jamsa

Professional DOS users and programmers with an interest in OS/2™, this book is a must for you! Jamsa, the author of the best-selling *DOS: The Complete Reference*, now shows experienced DOS users how to wield this operating system in powerful ways. If you're already familiar with C or Pascal, you'll gain even more insight from Jamsa's expertise. As a special highlight, the advanced features of DOS are compared with those of the new OS/2 operating system throughout the book. *DOS: Power User's Guide* shows you how to utilize fully the DOS pipe, memory map, system services, and subdirectories. Learn how to adapt the DOS environment to meet your specific needs. DOS pretender commands, disk layout, and system configuration are additional topics that Jamsa covers as he instructs you in the art of becoming a truly sophisticated user.

\$22.95 p

0-07-881310-7, 700 pp., 7³/₈ x 9¹/₄

UNIX®: The Complete Reference System V Release 3

by Stephen Coffin

The Complete Reference series now includes a book for all Unix® users! *UNIX®: The Complete Reference* includes expansive coverage of System V Release 3, the version that runs on 386 machines and on the new Macintosh™ II. Stephen Coffin approaches UNIX with a perspective that benefits users of microcomputers, minicomputers, and mainframes. There is special, timely coverage of UNIX on the 80386 micros; all code in the book was run on a 386. If you're just beginning UNIX, the first part of the book will help you get started. If you're an experienced UNIX user, this book is an invaluable and extensive reference. *UNIX®: The Complete Reference* offers discussions of commands, text processing, editing, programming, communications, the Shell, and the UNIX file system. Important highlights include running MS-DOS® under UNIX, upgrading to Release 3, and extensive coverage of UNIX on the 386.

\$27.95 p, Hardcover Edition

0-07-881333-6, 750 pp., 7³/₈ x 9¹/₄

\$24.95 p, Paperback Edition

0-07-881299-2, 750 pp., 7³/₈ x 9¹/₄

Order Today!

Call Toll-Free 800-227-0900

Use Your American Express, Visa, or MasterCard

Using the Model 50 & 60

by Herbert Schildt

Using the Model 50 & 60 is the best guide around to IBM's new machine in the Personal System/2™ series. A general introduction will start you off with a look at Model 60 components and an overview of the system. The book includes sections covering both DOS and OS/2 so that you'll have the information you need for whichever operating system you choose. Schildt discusses all essential OS/2 commands, as well as system configuration, multitasking, and advanced OS/2 features. If you're a DOS user, there's a comprehensive section written just for you, with an intro to DOS and a detailed description of DOS commands, system configuration, and advanced DOS features. *Using the Model 50 & 60* also covers all of the hardware—including 80286 specifics, memory, interrupts, disk drives, keyboard, video, and printers.

\$21.95 p

0-07-881308-5, 600 pp., 7³/₈ x 9¹/₄

Inside the Model 80

by Chris H. Pappas and William H. Murray, III

Serious programmers and software designers can take a look inside the IBM® PS/2™ Model 80 while exploring with Pappas and Murray, two 80386 masters. After a brief introduction to the Model 80 environment, Pappas and Murray discuss operating systems, DOS versus OS/2™, and consider the advantages and limitations of each. Next, the book deals with hardware features at the board and chip level, and with interfaces to disk drives, the monitor, mice, keyboards, and serial and parallel ports. As the authors of *80386/80286 Assembly Language Programming* and *80386 Microprocessor Handbook*, Pappas and Murray are uniquely qualified to write the chapters on assembly language techniques and applications. Specialized applications and workstation concepts, including desktop publishing, and CAD/CAM, are among the additional topics that are described in detail.

\$22.95 p

0-07-881311-5, 500 pp., 7³/₈ x 9¹/₄

Using QuickBASIC®

by Don Inman and Bob Albrecht

Here's an excellent programming guide to Microsoft's newest version of QuickBASIC™. And it's written by two programming professionals: Don Inman, a computer instructor and author of 16 computer books; and Bob Albrecht, the co-founder of DR. DOBB'S JOURNAL and a leader in computer education. Inman and Albrecht approach QuickBASIC's programming environment in three stages so beginning and experienced BASIC programmers can find the appropriate level of instruction. You can start with a thorough explanation of QuickBASIC's compatibility with IBM's BASICA, GWBASIC, and Applesoft™ BASIC; then you can move on to linking and running your BASIC code. Finally, you'll learn about subprograms, libraries, metacommands, and the important advantage of QuickBASIC's speedy graphics. You'll have a complete understanding of how this compiler works when

you finish *Using QuickBASIC®*.

\$19.95 p

0-07-881274-7, 325 pp., 7³/₈ x 9¹/₄

QuickBASIC™: The Complete Reference

by Steven Nameroff

QuickBASIC™: The Complete Reference is a comprehensive guide to Microsoft's extremely powerful QuickBASIC compiler. This resource is written for users at all levels of programming ability from novices to pros. Nameroff, coauthor of *Turbo Pascal® Programmer's Library*, has divided the book into sections to help you easily locate the information you need. *QuickBASIC™: The Complete Reference* begins with a quick introduction to BASIC programming, followed by a complete command reference section and a discussion of QuickBASIC functions, procedures, files, and graphics. Advanced techniques and applications are grouped together in the last section of the book for the professional QuickBASIC programmer. In addition, Nameroff considers the Microsoft CodeView® debugger and QuickBASIC compatibility with Microsoft® BASIC and BASICA. Nameroff's reference gives you all the answers to your questions on QuickBASIC, so keep it within easy reach.

\$27.95 p, Hardcover Edition

0-07-881363-8, 700 pp., 7³/₈ x 9¹/₄

\$24.95 p, Paperback Edition

0-07-881362-X, 700 pp., 7³/₈ x 9¹/₄

These titles available at fine book stores and computer stores everywhere

Or Call Toll-Free 800-227-0900

Use your American Express, Visa, or MasterCard

For a FREE catalog of all current publications, call 800-227-0900 or
write to Osborne/McGraw-Hill, 2600 Tenth Street, Berkeley, CA 94710

Prices subject to change without notice.

U S I N G

TURBO PROLOG

SECOND EDITION

"Using Turbo Prolog is good. Very good. It is probably as detailed and complete as any book about a computer program or language can be. I give this book the highest recommendation for anyone who has bought, or is contemplating buying, Borland's Turbo Prolog."
ONLINE TODAY

"... Worth hunting down before obtaining a Turbo Prolog system."
RYTE

Borland's new version 2 of the Turbo Prolog® language development system, designed for fifth-generation language programming and the creation of artificial intelligence systems, is explained for beginning and intermediate programmers in this excellent guide.

The newest edition of **Using Turbo Prolog®** is written with version 2 users in mind. While covering all the essentials of version 1, it explores version 2 enhancements, including its:

- New predicates and constants
- Ability to access external databases
- The Borland Graphics Interface™

Run and Rotate™ also thoroughly explain Turbo Prolog's essential:

- Statements, functions, and operations
- Multiple windows for viewing and modifying programs
- Color graphics and sound
- Techniques for creating control structures

Follow numerous examples and hands-on exercises, and you'll quickly go from fundamental procedures to advanced concepts.

An outstanding instruction guide and learning reference, **Using Turbo Prolog®**, Second Edition is your complete resource for the newest version of Borland's language development system.

Kelly M. Ricci is a Borland computer engineer and technical consultant and was a contributing editor for the Turbo Prolog user's group.

Phillip R. Robinson is a computer columnist for the San Jose Mercury News and the author of several books.

Borland Graphics International's Prolog™ and Turbo Prolog is a registered trademark of Borland International, Inc.

ISBN 0-07-881302-6

019-95